PREDICTABLE AND MONITORED EXECUTION FOR COTS-BASED REAL-TIME EMBEDDED SYSTEMS

BY

RODOLFO PELLIZZONI

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2010

Urbana, Illinois

Doctoral Committee:

Professor Lui Sha, Chair
Associate Professor Marco Caccamo, Director of Research
Assistant Professor Deming Chen
Professor Sanjoy Baruah, University of North Carolina

Abstract

Modern real-time embedded systems are moving from federated architectures, where logical applications and subsystems are implemented on different hardware components, to progressively more integrated architectures which use extensive sharing of different physical resources. These systems employ multiple active components, such as CPU cores, HW processors, coprocessors and peripherals, which can all autonomously perform computational and communication activities. Furthermore, they are increasingly built using Commercial Off-The-Shelf (COTS) components in an attempt to increase performance and reduce cost and time to market.

Integrated real-time systems such as those employed in the avionic, medical and automotive domain are often *mixed-criticality* systems: they implement different applications with widely varying levels of criticality. Therefore, a key issue is to provide sufficient isolation among different applications. In particular, safety-critical applications can expose requirements both in terms of *functional isolation*, e.g. fault containment, and in terms of *physical isolation*, e.g. safe sharing of physical resources such as CPU and communication time, memory and power.

In this work, we study the design of mechanisms and policies to support both functional and physical isolation, with a special focus on timing guarantees. In particular, since most available COTS components do not provide sufficient hardware isolation mechanisms, we propose the concept of a *control abstraction*: an unintrusive hardware device or software layer that is interposed between a COTS component and the rest of the system, allowing the system architect to predictably control all its resource accesses. By employing control abstractions, unverified COTS components can be used to implement low-criticality but high-performance applications, while still providing all required isolation guarantees to safety-critical modules. Functional isolation is provided by monitoring the run-time communication behavior of the component against a formal specification, and taking a recovery action whenever the specification is violated. Timing isolation is provided by coscheduling all computational and communication activities in such as way that there is no contention for access to system resources.

We show the validity of our methodology by applying it to two different embedded architectures. For System-on-Chip architectures, we detail a complete platform-based design process that automatically generates control abstractions for all integrated processors from a high-level functional system specification. We test the described design process on the case study of a medical pacemaker.

For COTS-based computational nodes, we focus on the contention between CPU tasks and peripherals for access

both to shared communication infrastructures such as PCI and to main memory. Our experiments show that main memory interference can greatly increase the worst-case execution time of a task, up to almost 200% for a dual core system with a single PCIe peripheral. To overcome this issue, we propose both analysis techniques to compute upper bounds on the worst-case task delay, as well as hardware and software control abstractions to reduce such delay. In particular, we detail the design and implementation of a new hardware device, the *real-time bridge*, which is interposed between each COTS peripheral and the PCI bus. The real-time bridge buffers all incoming/outgoing traffic to/from the peripheral, and delivers it predictably according to a defined schedule. Furthermore, we propose to execute CPU tasks according to a new PRedictable Execution Model (PREM), which uses a combination of compiler techniques and OS modifications to precisely control all main memory accesses performed by a task. By combining PREM with the real-time bridge, we can coschedule all accesses in main memory by both peripherals and tasks, thus eliminating low-level contention and unpredictable access delays. Our experiments show reductions in worst-case execution time up to 40%-60% compared to a traditional execution model.

Table of Contents

Chapter 1

Introduction

The real-time and embedded systems communities have accomplished major milestones in the last three decades that have profoundly changed the way we design embedded systems and think about them. However, real-time resource management is currently facing formidable challenges due to a paradigm shift in the way complex embedded systems are developed: we are moving from federated systems, where logical applications and subsystems are implemented on different hardware components, to progressively more integrated architectures which use extensive sharing of different physical resources such as CPU time, chip area, and power. This trend has dominated the general consumer market for at least the last decade, but is now also emerging in the area of safety-critical systems. Most of the innovation factor in the automobile market depends on software functionalities rather than improvements in mechanical engineering. The medical device market is subject to a strong push towards the implementation of advanced interoperability functionalities (medical plug and play [1]) to ease system assembling and usage and reduce the risk of human error.

When safety critical embedded systems are implemented based on an integrated architecture, different applications sharing the same physical resources can have widely varying levels of criticality. For example, in modern cars a critical system such as the anti-lock braking system (ABS) and a non critical system such as information display share basic sensors (tachometers). Current pacemakers provide advanced logging and reporting mechanisms together with the life-critical heart pacing functionality [89, 103, 10]. A key issue in these *mixed-criticality* systems is to provide sufficient isolation to different functional partitions: safety-critical applications are typically simple enough that they can be formally verified and/or certified through testing[1], but less critical applications are often complex enough that certification is completely unfeasible from both a timing and monetary perspective. As such, it is essential to ensure system isolation, so that faults in a less critical hardware or software component will not affect a more critical one. In more details, I believe that we can distinguish between two types of isolation, both required for the correct and safe operation of a system.

[1] In general formal verification is carried out on a model of the system and it can not ensure that the actual implementation is correct. This is the reason why, for example, the Federal Aviation Administration requires stringent testing before certification is issued for any flying vehicle.

- *Functional isolation*, or fault containment, expresses the well-understood property that erroneous results produced by one application should not affect the results of computation of a second application; in other words, functional isolation depends on the exchange of logical data between applications.

- *Physical isolation* expresses the property that faults in one component can not cause it to interfere with the shared resource usage of a second component, preventing it from producing expected results. Physical isolation encapsulates different requirements, because integrated embedded systems include different types of shared resources, in particular: **(1)** CPU and communication resources can typically be partitioned in the time domain, e.g. the resource can be scheduled. This is the domain of real-time scheduling theory, and we refer to its associated requirements as *timing isolation*. **(2)** Memory, cache and buffer space can be statically or dynamically partitioned among applications. **(3)** Finally, power is an important shared resource in battery operated device, and no hardware component should be allowed to disproportionally increase power consumption reducing the system lifetime.

In this thesis, I will explore architectural solutions and design methodologies to provide strong functional and physical isolation guarantees for modern safety-critical and mixed-criticality embedded systems. In terms of physical isolation, I will mostly focus on timing properties, but memory and power allocation will also be discussed in specific architectures. In general, the isolation problem is not new, in the sense that solutions are available for architectures such as traditional CPUs. In particular, a basic level of functional isolation is provided by hardware mechanisms such as virtual memory. These mechanisms can further be used to provide some level of physical isolation on top of functional isolation: hardware privilege levels prohibit a low criticality partition from tempering with the OS, which can in turn provide real-time services to guarantee timing isolation. Unfortunately, as I detail in the following motivating examples, modern mixed-criticality embedded systems are complex interconnected systems, and CPU protection alone is not sufficient to provide isolation guarantees.

Example 1: COTS-based architecture in avionic systems. Avionic systems are increasingly built by using Commercial Off-The-Shelf (COTS) components in an attempt to reduce costs and time-to-market [11]: it is becoming difficult to rely on completely specialized hardware and software solutions since performance is much lower when compared to equivalent COTS components developed for the mass market. For example, the specialized SAFEbus backplane [43] used in the Boing777 is capable of transferring data up to 60 Mbps, while a modern COTS interconnection such as PCI Express 2.0 [69], which is found in common Personal Computers (PC), can reach transfer speeds over three orders of magnitude greater at 16 Gbyte/s. This performance improvement is relevant, because advanced avionic sensors produce high amount of data traffic that must be processed by the system. Similarly, speculative CPU cores with deep pipelines and multi-level cache hierarchies are used to provide the level of computational performance needed to process and aggregate all sensor inputs. Unfortunately, integrating COTS components in safety-critical systems is extremely challenging. High performance peripherals can perform a variety of specialized computational tasks, like packet processing in Network Interface Cards (NICs), and can use Direct Memory Access (DMA) mode to

directly initiate read/write transactions towards either other peripherals or main memory. CPU clock is typically much faster than memory speed, and any cache miss in last level cache can easily empty the pipeline and stall a core while waiting for the required cache line to be fetched from main memory. However, most COTS architectures still feature main memory organized in one or multiple single-port banks that are shared among all peripherals and CPU cores. When a task suffers a cache miss, contention for main memory access can significantly delay cache line fetch, greatly increasing the worst case execution time of the task. In the experiments detailed in Section 4.4.4, we show that in a dual-core system with a single PCI Express peripheral, task computation time can be increased by up to 196% in the worst case.

Example 2: heterogeneous SoC platform. Systems-on-Chip are rapidly becoming a major driving force behind the growth of the embedded system market. By combining all or most system components on a single silicon chip, SoC designs usually consume less power and have a lower cost and higher reliability than the multi-chip systems they replace. To benefit from economy of scale, several semiconductor companies have began to offer standardized SoC platforms that, while targeted at a specific sector of the market, can be customized and reused for a variety of applications. Most platforms are widely heterogenous, incorporating a variety of different processing components. For example, the Cell Broadband Engine [23] features a general purpose PowerPC core and eight Synergistic Processing Elements, which are high-frequency in-order cores especially designed for multimedia applications; all cores share a common communication infrastructure and memory. The recent QorIQ PowerPC platform developed by Freescale [37] integrates multiple cores together with high speed communication elements on the same chip, and is especially marketed for critical avionic and military applications. In general, SoC platforms make large use of COTS Intellectual Property (IP) blocks (such as existing CPUs, coprocessors, etc.) to simplify development and reduce time-to-market. Customization can be provided by either software processors (general purpose CPU or DSP executing software code) or customizable hardware processors implemented on reconfigurable logic (FPGA) [110]. All processing elements are active components, able to initiate read/write transactions on the communication infrastructure and automatically retrieve/push needed data from/to the network.

The two proposed examples share two fundamental characteristics, which are indeed true of most modern embedded architectures: **(1)** the system is composed of multiple *active components*, such as CPU cores, HW processors, coprocessors and peripherals. These active components autonomously perform computational activities, but must communicate to collaboratively execute an application. **(2)** There are multiple shared resources in the system. CPU computation time is just one among them; multiple active components contend for access to communication elements, such as buses and network-on-chip routers, as well as external memories, and cache and other memories are shared among multiple applications.

Therefore, in this work I study the design of isolation mechanisms for COTS-based embedded systems featuring multiple active components and shared resources. In particular, system evaluation and prototyping efforts are focused on the discussed motivating examples, e.g. COTS-based PC systems and heterogeneous SoC platforms. Compared to

traditional work in the real-time and embedded system area, these emergent architectures present several challenges:

- COTS components are inherently unpredictable: they are designed to improve average-case performance, not worst-case performance as required in safety-critical systems. In particular, the low-level arbiters used by COTS components to regulate access to shared resources do not typically support timing-predictable sharing schemes: this can cause long waiting times in the worst case. Furthermore, COTS standards are often imprecise and/or incomplete when it comes to timing aspects; specifications are drafted by manufacturers, which are interested in being able to optimize performance. To mitigate these issues, it is necessary to regulate the operating point of each COTS shared resource and maintain it below saturation limit.

- There is a lack of standardized low-level hardware control mechanisms. As previously detailed, logical isolation among software partitions running on the same core is possible because the CPU provides suitable hardware protection mechanisms such as virtual memory and privilege levels. Unfortunately, other active components do not offer the same level of hardware protection. To the best of my knowledge, there is no standardized protection mechanism implemented for HW coprocessors in any commercially available SoC platform. COTS peripherals are particularly problematic, since they can potentially access shared communication elements and main memory at any time. A typical high-performance NIC performs a transfer in main memory whenever it receives a data packet, and it is very hard to control exact packet arrival times in a large distributed system such as an aircraft. Furthermore, a fault in the driver or in the peripheral hardware can potentially compromise any other application in the PC node.

- Scheduling computational activities is not enough to ensure timing isolation. Since multiple heterogeneous active components must collaborate to execute an application, computation can be strongly influenced by communication aspects. Therefore, to achieve timing predictability, computational and communication activities should be *coscheduled* in the system. Furthermore, note that scheduling results obtained in the area of distributed system are not necessarily well suited or even applicable to embedded nodes: compared to classic distributed systems, our systems of interest are much more tightly integrated. In particular, communication tends to be based on homogeneous infrastructures and synchronization delays are shorter. On the other hand, most results about distributed systems assume a completely asynchronous model of computation, which has significant drawbacks in terms of complexity of reasoning about system properties.

To address these challenges, several research projects [28, 3, 104] are proposing solutions based on the design of new high-performance hardware components with built-in isolation features. For example, the PREcision Timed (PRET) machine [28] is a pipelined 6 core system, where each instruction takes a deterministic amount of time to be executed. All sources of unpredictability in the CPU architecture, such as caches and branch prediction, are systematically substituted by predictable mechanisms such as scratchpad memories and compiler-provided hints. Furthermore, all communication resources in the system are accessed based on a TDMA scheme that is synchronized with the CPU

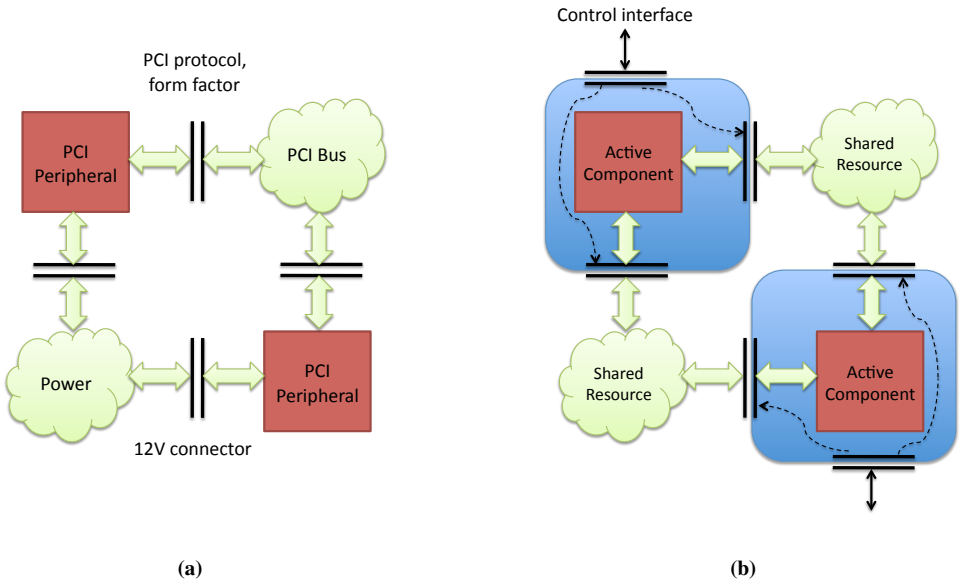


Figure 1.1: A COTS-based system **(a)** without and **(b)** with control abstractions.

clock. These architectural features make it possible to derive tight static analysis bounds on task worst-case execution time. Unfortunately this approach, as well as similar work in the literature, presents in my opinion some applicability issues: **(1)** it requires a complete redesign of the hardware architecture. In the case of PRET, this includes the CPU cores, the memory subsystem, the communication infrastructure and all peripherals. This makes it extremely difficult to reuse available IPs, and can not take advantage of the economy of scale that is driving performance improvements in the general purpose market. **(2)** In several cases, it requires recoding of all software applications, or at least a redesign of the compiler infrastructure. Especially in the case of mixed-criticality systems, industry often prefers to follow some form of software migration path, which is perceived as less risky. **(3)** Safety typically depends on the correctness of all hardware components, which individually enforce scheduling and functional isolation constraints. Therefore, all system components must be provably safe. This is problematic in mixed-criticality system, where it would be preferable to only verify the hardware components used to run critical applications.

In this work, I explore a different approach: instead of redesigning all hardware devices, I propose to reuse available COTS hardware and software components. The required logical and physical isolation properties can be guaranteed through a combination of new architectural features and design techniques. To provide strong isolation guarantees, it is first necessary to develop suitable low-level control mechanisms that are able to reign in the unpredictability of COTS components. In Chapter 2, I first propose the concept of *hardware control abstraction*, an unintrusive hardware device that can be seamlessly integrated in COTS-based embedded systems. Its role is to *abstract away* the unpredictable low-level implementation details of a COTS component, allowing the system architect to *predictably control* all its resource accesses. A high-level schematic representation of the idea behind control abstractions is represented

in Figure 1.1, where active components access shared system resources through standardized interfaces, which include both physical aspects (such as the form factor of a PCI peripheral board) and functional aspects (such as the PCI protocol used to communicate and initialize the peripheral). Hardware control abstractions act as wrappers around each active component: they export an additional interface that can be used to control the other interfaces accesses by the component, and therefore the way the component uses system resources. Based on the control functionalities exported by control abstraction, in Chapter 3 I then detail my solution to provide functional isolation in mixed-criticality systems. Instead of formally verifying and/or certifying a COTS component executing a low-criticality application, we integrate *monitoring* functionality in the control mechanism: at run-time, the behavior of each COTS component is formally checked. If the component is found to be misbehaving, the hardware control abstraction can be used to prevent damage to high-criticality applications. Finally, Chapters 4 and 5 are dedicated to timing isolation. As described in our first motivating example, contention for access to main memory is a main source of unpredictability in modern COTS-based systems. In Chapter 4, we detail a set of analysis methodologies to derive safe upper bound on the maximum delay that a task can suffer due to memory contention interference. Unfortunately, analysis bounds do not scale well when the number of CPU cores or peripheral interconnections in the system increases; new engineering solutions are required to minimize contention delay. In Chapter 5, I show how control abstractions can be used to schedule all active components in the system, eliminating resource contention at the level of unpredictable COTS arbiters. In the remaining of this section, I provide a more detailed overview of the content of each chapter to give a better picture of the overall scope of this work.

When developing hardware control abstractions, a key consideration should be kept in mind: the control mechanism must be compatible and work transparently with respect to the interfaces used by COTS components. In Chapter 2, I present two different examples of control abstraction design for different embedded architectures: an I/O management system for peripheral traffic in a standard PC architecture in Section 2.1, and a SoC platform for mixed-criticality systems in Section 2.2. The described solutions differ in the amount and quality of the modifications that can be introduced in the base system. For PC systems, it is critical for the introduced control mechanisms to be compatible with the complex PCI standard from a protocol and physical point of view. Our I/O management system adds two new hardware components to the system. A *real-time bridge* is interposed between each COTS peripheral and the rest of the system. The real-time bridge buffers all incoming/outgoing traffic to/from the peripheral, and delivers it predictably according to assigned timing reservations. The *peripheral scheduler* uses the control interfaces exported by all real-time bridges and computes the actual transmission schedule based on the timing reservations. Real-time bridges are implemented as complete system-on-chip solutions based on a standard OS and peripheral drivers. On the other hand, SoC design provides more freedom: in particular, our platform reuses COTS components for the communication infrastructure and software processors (CPUs). However, each active component in the system is encapsulated in a specially-designed *monitoring hardware wrapper* which controls all communication and resource usage by that component. In particular, in Section 2.2 we detail a complete platform-based design methodology that

automatically generates all hardware wrappers from a high-level functional specification of the system. In both cases, the implemented control mechanisms provide similar services: they are capable of stopping a master from accessing system resources, either temporary to perform scheduling or correct a transitory fault, or permanently in the case of a non-recoverable failure. System resources include the communication infrastructure and, for battery operated SoC systems, power consumption: since we assume that all active components communicate through a unique communication infrastructure, stopping access to the infrastructure will effectively prevent a component from interfering with the rest of the system.

This base service is exploited to provide functional isolation in Chapter 3. As previously mentioned, my approach to functional isolation is based on the concept of runtime monitoring: requirement specification for each active component is checked at runtime against its current observable behavior. If any violation is detected, then a suitable recovery action can be taken to restore the system to a safe state. Runtime monitoring can greatly simplify validation and certification of COTS components. Instead of verifying the complex inner working of each device, we more simply need to prove that: **(1)** the system is kept in safe state as long as the requirement specification for all COTS active components is met, and **(2)** recovery actions can correctly restore the system to a previous safe state if a requirement is violated. The validity of the runtime monitoring approach has been proven in the field of software engineering by a large number of developed tools and techniques. In particular, I leverage on the Monitor Oriented Programming (MOP) [22] framework developed by Prof. Grigore Rosu at UIUC, which is highly extensible and supports multiple formalisms. MOP had previously been applied only to check the correctness of software applications. In this work, I propose to extend it to cover hardware modules as well. This is not trivial, since it requires both suitable hardware mechanisms to perform fast run-time checks, and new methodologies to formally describe the hardware requirement specification. The applicability of our technique is proven through two case studies, involving a PCI data acquisition card and the SoC design of a medical pacemaker.

The main contribution of Chapter 4 is the development of a theoretical framework able to model the interaction between CPU cores and peripherals contending for shared main memory and to derive safe upper bounds on task worst case execution time. The analysis is based on a characterization of both the peripheral traffic and the task memory accesses. In particular, we assume that burstiness bounds are provided for peripherals, and a characterization of suffered cache misses over time, called a *cache profile*, is available for each real-time task. Since deriving the cache profile can be challenging, we provide analyses based on two different profile models: the exact model requires a precise trace of cache misses, while the interval-based model only assumes that the maximum number of misses is known for a given time interval. The obtained execution time bounds are tight for single core systems. The analysis for multicore systems is more challenging, because cache fetches and write-backs performed on one core can delay cache fetches on a second core. In particular, we show how to derive a characterization of the worst-case memory traffic generated by the first core based on the tasks it executes. In this case the derived execution time bounds are not tight, but they are fairly close to the real worst-case.

While the analysis in Chapter 4 provides safe bounds on the worst case execution time of a task due to memory interference, the worst case interference delay can be very large, especially for multicore systems. Based on the hardware control abstractions introduced in Chapter 2 and the delay analysis of Chapter 4, in Chapter 5 we show how task delay can be minimized by coscheduling CPU tasks and I/O transactions. In particular, we describe two different solutions, based on the task execution model. When a typical task executes, cache misses tend to be highly unpredictable. In Section 5.1, we describe a run-time coscheduling heuristic for single core systems. Each task is assigned an offline computation time budget that is sufficient to execute the task assuming no memory interference from peripherals. The run-time heuristic then maximizes the traffic transmitted by all peripherals, while guaranteeing that each task still completes within its assigned time budget. Unfortunately, in the worst case no peripheral traffic will be allowed, which makes it impossible to provide guarantees on I/O flows. Our PRedictable Execution Model (PREM), detailed in Section 5.2, solves this issue. PREM uses a combination of compiler techniques and OS modifications to precisely control all main memory accesses performed by a task; effectively, PREM acts as a software control abstraction for tasks running on an unmodified COTS CPU with multiple cache levels. Since using PREM we can predictably control resource accesses by all active components in the system, a much improved coscheduling algorithm can be employed, where no two active components access main memory at the same time. Hence, no low-level contention for resource access is possible, and neither tasks nor peripherals can suffer unpredictable delays. Finally, our implemented single core prototype shows that PREM incurs only limited overhead compared to traditional execution.

1.1 References and Acknowledgements

The present dissertation is partially based on material previously published in peer-reviewed conferences and journals, in particular [9, 75, 74, 72, 76, 70]. This work would not have been possible without the contribution of all coauthors of the referenced publications. The most important contributor has undoubtedly been my advisor Prof. Marco Caccamo. Marco's mentoring has been essential to establish my research vision. I am extremely grateful to him for providing an outstanding level of feedback, for believing in my ideas and for involving me in all aspects of running a successful research agenda. A special thanks goes to Prof. Lui Sha for his precious advices and for being an incredible source of new, excellent research ideas. A significant portion of the original ideas contained in this work derives from conversations with Prof. Sha.

I am also grateful to Prof. Deming Chen and Prof. Sanjoy Baruah for serving on my thesis committee and for their kind comments, and to Prof. Grigore Rosu, Prof. Lothar Thiele and Prof. Jian-Jia Chen for the fruitful research collaborations that we established over the past few years; their expertise and advices have been fundamental to extend my work in new directions. Similarly, the impact and applicability of this dissertation would be significantly lower without the precious insights provided by our industrial partners.

Last but not least, I would like to thank the following fellow PhD students for their contributions to selected portions of this work, and more in general for the many interesting research discussions over the past five years.

- Emiliano Betti for his precious work on kernel and driver development, in particular in relation to Sections 2.1.2 and 5.2.4.
- Stanley Bak for the hardware implementation of the peripheral scheduler in Section 2.1.1.
- Both Emiliano Betti and Stanley Bak for their contribution in obtaining the bus traces shown in Sections 2.1.4, 5.2.4.
- Patrick Meredith for his help in defining and implementing the HardwareMOP language in Section 3.3.
- Andreas Schranzhofer for the arrival curve derivation in Section 4.4.1.
- Min-Young Nam for the AADL tool in Section 2.2.2.
- Gang Yao for the PREM compiler implementation in Section 5.2.4.

Chapter 2

Hardware Control Abstractions

As detailed in Chapter 1, the concept of control abstraction is key to our isolation methodology: instead of redesigning hardware components to natively support properties of functional and physical isolation, we "wrap" each COTS component in a suitable hardware device which controls all interactions of the component with the rest of the system and enforces the required isolation properties. In this way, existing hardware IPs can be reused in a mixed-criticality system.

In general, implementing control abstractions is not easy. In particular, a good abstraction design should strive to satisfy several properties. **(1)** The abstraction should not require any change in the COTS interfaces used by the controlled component to connect with the rest of the system. Ideally, the control abstraction should be completely transparent: from a functional point of view, the rest of the system should be able to communicate with the controlled component as if the abstraction were not present. **(2)** The abstraction should be able to control access to all shared resources used by the component. **(3)** The abstraction should not significantly degrade the performances of the controlled component. COTS components are especially attractive for their performance advantage over specialized solutions. Such advantage should not be lost. **(4)** The abstraction should have low overhead in term of resources, such as power, chip area, weight, etc., required for its control mechanisms. **(5)** The implementation of the control mechanisms should be as simple as possible. Integrating COTS components in mixed-criticality systems is challenging because COTS components are complex to verify; hardware abstractions should be easier to verify than their controlled components. **(6)** The implemented control mechanisms should be reusable as much as possible over a large class of controlled components. Hardware abstractions are clearly dependent on the controlled shared resources and interfaces, therefore a different abstraction must be developed for each COTS architectural standard. However, it is important for the abstraction to be reusable for all components adhering to a given standard. **(7)** Abstraction deployment should be automated as much as possible. Hardware abstractions provide a way to control access to system resources, but do not mandate any specific control policy. It is important to develop suitable analysis and design methodologies to help the system designer devise the best control solution for his system.

In this chapter, we discuss the implementation of two concrete hardware control abstractions and their associated design methodologies. The I/O management system in Section 2.1 uses a real-time bridge to control bus traffic generated by PCI peripherals. In Section 2.2, we show a platform-based design approach for mixed-criticality SoC systems, where hardware wrappers are used to control access to communication resources and power consumption. The employed control mechanisms are very different in the amount of constraints imposed on them by the controlled architecture, and we believe that they are fairly representative of the range of solutions possible in COTS systems. In Section 2.3 we will then discuss how each control abstraction fares in terms of the desirable properties listed above and provide concluding remarks.

2.1 I/O Management System for COTS Peripherals

A COTS PC system typically includes several commercial peripherals, such as video acquisition boards or network cards, plugged into standard slots on a commercial motherboard. The Peripheral Component Interconnect (PCI) [69] is the current standard family of communication architectures for motherboard/peripheral interconnection in the PC market; it is also widely popular in the embedded domain [87]. The standard can be divided in two parts: a *logical* specification, which details how the CPU configures and accesses peripherals through the system controller, and a *physical* specification, which details how peripherals are connected to and communicate with the motherboard. The logical specification has remained largely unaltered since its introduction, while several different physical specifications are available: the original PCI and PCI-X architectures use shared bus segments connected by bridges, while the new PCI Express (PCIe) specification is based on point-to-point serial links between peripherals and PCIe switches. In both cases, data from PCI peripherals travels through a series of bridges/switches, until it reaches main memory, where the CPU can typically read it through a dedicated interconnection such as the Front Side Bus (FSB). Alternatively, the CPU can write data into main memory and instruct the COTS peripherals to retrieve it. For example, a network card could be instructed to upload packets which are stored in RAM. I/O components such as these can inject significant traffic onto the I/O bus of modern real-time systems. For example, a search and rescue helicopter [78] may include several high-bandwidth components such as a Doppler navigation system, a forward look-ahead infrared radar, a night vision system, and several types of communication systems. A single real-time high-definition video may consume an I/O bandwidth of tens to hundreds of Mbps [5].

While priority-based real-time scheduling is a standard practice for the CPU, it is currently not supported by COTS peripherals and interconnect systems such as PCI. Due to the lack of real-time prioritization, data I/O transactions traveling through the COTS bus into or out of main memory can suffer unpredictable delay and cause deadline misses [65]. We address this challenge by introducing a real-time I/O management system that supports a wide range of priority-based scheduling policies, retains backward compatibility with existing COTS-based components, and achieves high real-time bus utilization without degrading peripherals' throughput. In particular, our control mechanism provides the following novel features: 1) it enforces predictable bandwidth reservations for I/O COTS peripherals, 2) it does

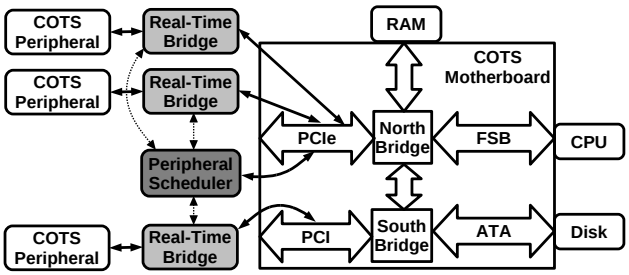


Figure 2.1: I/O Management System.

not require synchronization between the CPU scheduler and the I/O subsystem, and 3) it facilitates lossless reshaping (under given assumptions) of bursty traffic from a network of distributed real-time nodes. Finally, as required for an hardware control abstraction, our management system is completely transparent to CPU applications: in fact, as detailed in Section 2.1.2, we were able to use a network card through our prototype without any user-end application modifications.

The real-time I/O management system, shown in context in Figure 2.1, adds two types of components to the existing COTS system. A *real-time bridge*, our hardware control abstraction, is interposed between each peripheral and the communication bus. Each real-time bridge provides the actuation mechanism to enforce peripheral bus access. The second component is the *peripheral scheduler*, which implements the system-wide policy for accessing the bus. It can be thought of as a high-level arbiter which instructs the real-time bridges to either communicate on the bus, or yield to other devices. We implemented both of these components in hardware on field programmable gate arrays (FPGAs), although an industrial application would likely use an Application Specific Integrated Circuit (ASIC). We describe these devices in Section 2.1.1 and Section 2.1.2 and elaborate on details involved in making the prototypes. In Section 2.1.3 we then show how delay and required buffer sizes for I/O flows can be computed based on Real-Time Calculus theory [16]. In Section 2.1.4 we demonstrate, on physical hardware, that timing violations can occur without our real-time I/O management system, and that our control mechanism can prevent I/O deadline misses. Finally, Section 2.1.5 discusses related work.

2.1.1 Peripheral Scheduler

Multiple peripherals must cooperate to prevent a timeliness reduction caused by mutual interference. The peripheral scheduler centralizes decision making and coordinates multiple real-time bridges by instructing them to either forward or buffer peripheral traffic. Presently, we consider each peripheral as generating a single real-time I/O flow. In Section 2.1.2, we describe the changes necessary to permit a single peripheral to send multiple real-time I/O flows with potentially different priorities. Furthermore, we only consider I/O flows directed to main memory, and not to other

Code 1 These logical expressions, implemented in hardware on the peripheral scheduler, provide a static-priority I/O scheduler for a four-peripheral system.

$\texttt{block}_0 := \neg(\texttt{data\_rdy}_0)$

$\texttt{block}_1 := \neg(\texttt{block}_0 \wedge \texttt{data\_rdy}_1)$

$\texttt{block}_2 := \neg(\texttt{block}_0 \wedge \texttt{block}_1 \wedge \texttt{data\_rdy}_2)$

$\texttt{block}_3 := \neg(\texttt{block}_0 \wedge \texttt{block}_1 \wedge \texttt{block}_2 \wedge \texttt{data\_rdy}_3)$

peripherals[1]. As described in the chapter introduction, and shown later in our evaluation in Section 2.1.4, allowing multiple COTS peripherals to simultaneously access main memory can result in unpredictable bandwidth allocation. As such, we allow only a single real-time bridge to transmit at any one time. Therefore, we can consider the time allocated among all real-time bridges by the peripheral scheduler as a shared resource akin to a monoprocessor CPU. In this analogy, each I/O flow is equivalent to a real-time task, and each I/O data chunk in the flow is equivalent to a job; transfer times for I/O data chunks are equated to computation times and can typically be derived by dividing the I/O data amount by the achievable throughput of the real-time bridge.

The control interface between each real-time bridge and the peripheral scheduler comprises two physical wires. Each real-time bridge communicates one boolean value, `data_rdy`, to the peripheral scheduler. This value indicates that data is buffered and ready to be sent on the bus. In turn, the peripheral scheduler sends one boolean value back to the real-time bridge, `block`, which instructs the bridge to either block I/O traffic or permit bus access. Real-time bridges instructed to block do not attempt to gain access to the bus, mandating the bus arbiter grants the unblocked peripherals access to the bus to send their data.

With only these two signals, many kinds of bus scheduling policies can be enforced. Consider, for example, scheduling four real-time bridges according to a static-priority bus scheduling scheme, such as rate monotonic (RM). Let $\texttt{block}_i$ be the block command sent to the ith real-time bridge, and let $\texttt{data\_rdy}_i$ be the indicator of buffered data coming from the ith real-time bridge. Let the bridges be physically connected to the peripheral scheduler in the order of their priorities (in the order of their rates for RM), from the highest priority bridge, $i = 0$, to the lowest priority bridge, $i = 3$. In order to provide static-priority scheduling on the I/O bus, the peripheral scheduler hardware would implement the logical expressions in Code Block 1.

Our framework, however, can also support a large class of monoprocessor scheduling algorithms which handle sporadic and aperiodic tasks using real-time servers [19]. In our prototype, for instance, we have implemented support for sporadic servers [8] under fixed-priority scheduling. Notice that the servers are implemented on the peripheral scheduler and not on the associated real-time bridges. This decision has two major advantages. First, it removes the need for precise clock synchronization among real-time bridges and the peripheral scheduler. Since all scheduling servers use the same physical clock, server budgets can be precisely calculated without clock skew. Second, it simplifies the interface between each real-time bridge and the peripheral scheduler by reducing the number of physical

[1] Our methodology will be extended to cover inter-peripheral communication in our future work.

wires to just two, `data_rdy` and `block`. This becomes a concern for algorithms like EDF, where each server must communicate a precise deadline timestamp to the scheduling algorithm, which requires dozens of bits of information. By centralizing all scheduling servers on the peripheral scheduler, the number of physical wires is reduced, simplifying the electrical design.

We now describe the way in which our proposed architecture can be used to provide real-time guarantees for peripheral traffic. Particularly, we are interested in determining the classes of schedulers that can be implemented using multiple real-time bridges connected to a peripheral scheduler. We address the following question: can any real-time monoprocessor scheduling algorithm be used with our hardware framework? Although we show that the answer is no, we are still able to employ a large class of monoprocessor schedulers. Since there are many classifications of scheduling algorithms, we start by showing a definition that provides a more formal framework on which to reason about implementable schedulers.

Definition 1 (Scheduling Servers / Scheduling Algorithms) *Consider a set of tasks requesting access to a single shared resource. Each task (or group of tasks) has an associated* scheduling server *which, based on the task's (tasks') activity, forwards scheduling parameters to the* scheduling algorithm*. At any time instant t, the scheduling algorithm grants the shared resource to at most one scheduling server based on the value of all received scheduling parameters.*

The provided definition is general enough to encompass all common real-time schedulers for a single shared resource. For example, consider scheduling periodic tasks according to rate monotonic. The scheduling server for each task forwards a STATIC_PRIORITY parameter equal to the inverse of the task period, and a READY boolean parameter which is true if and only if the task has remaining computation time. The fixed priority scheduling algorithm then selects the task with highest STATIC_PRIORITY and a true READY parameter to execute. A budget-based server for aperiodic tasks, such as a sporadic server, would be more complex, but would still output the same STATIC_PRIORITY and READY parameters. In particular, the server's READY parameter is true if and only if the served task has remaining execution and there is available budget. However, the set of scheduling parameters changes based on the scheduling algorithm. For instance, an EDF server would need to output both a READY boolean value and a dynamic ABSOLUTE_DEADLINE parameter.

The logical division between the scheduling servers and the scheduling algorithm corresponds to the physical implementation within the peripheral scheduler. Each scheduling server is implemented by a single VHDL hardware module. The module receives the `data_rdy` signal from the corresponding real-time bridge and computes the scheduling parameters (the boolean READY value and the STATIC_PRIORITY value). Global scheduling logic, such as the logic shown in Code Block 1, then implements the scheduling algorithm, receiving scheduling parameters and producing `block` signals for all scheduling servers (however, in the case of fixed priority scheduling, STATIC_PRIORITY is a design-time parameter which is absorbed in the implementation of the global scheduling logic as an optimization).

The main drawback of such a design, however, is that each scheduling server must make scheduling decisions based only on the task's `data_rdy` (active) value.

Definition 2 (Active-Dynamic Server) *A scheduling server is an* active-dynamic server *if run-time task behavior can be governed using only one piece of task-related knowledge, whether the task is active (immediately executable) or not.*

Proposition 1 *Our I/O scheduling mechanism can implement any scheduling framework where all scheduling servers are active-dynamic servers.*

Next, we provide some examples of servers that are active-dynamic, and some examples of servers that are not.

Lemma 1 *Under fixed priority scheduling, both a periodic task server and the sporadic server are active-dynamic servers.*

Proof.

A scheduling server servicing a periodic task needs only output the STATIC_PRIORITY for the task, and a dynamic READY parameter which is equal to the active value of the task.

A sporadic server is active-dynamic because the rules governing its replenishment time and replenishment amount can be obeyed knowing only the task's active value (and other non-task parameters such as the current time). According to the rules of a sporadic server, the replenishment time is determined as soon as the task becomes active (an aperiodic task requests execution) and there is available budget. The replenishment time is computed by simply summing the current time with the static server period. The replenishment amount, computed when the server becomes inactive, is equal to the consumed budget. This is computed by measuring the duration of time the task was both active and unblocked (which can be given through feedback from the scheduling algorithm). Determining when the server becomes inactive is done by checking if the task is finished or the budget is exhausted. Since all the server rules can be evaluated based only on the active status of the task, a sporadic server is an active-dynamic server. □

Lemma 2 *Consider a sporadic task feasibly scheduled under EDF with relative deadline greater than its interarrival time. The server for such task is not an active-dynamic server.*

Proof.

Consider two successive jobs j and $j+1$ of the sporadic task. Since the interarrival time between jobs is less than the relative deadline, job $j+1$ could arrive in the system before job j is finished. Since the server has no way to know when $j+1$ arrived based on the active status of the task, it can not set the deadline for job $j+1$. □

Also notice that there exist non budget-based servers, such as the total bandwidth server[95], which require arrival-time information about aperiodic job execution time and therefore are not active-dynamic servers.

Two additional architectural attributes need to be considered when developing the proposed real-time I/O management system. First, the I/O transfer time is only valid if the device receives the achievable throughput on the front side bus (FSB). Since the CPU main memory access is not regulated by the peripheral scheduler, it may concurrently access main memory. Second, for performance reasons, typical COTS buses do not allow individual bus transactions to be preempted and so they must run always to completion. This means that although we may activate a real-time bridge's `block` signal, the bridge will still need to complete its current transaction before relinquishing control of the bus, which may affect the schedulability. We address these concerns in order.

The first concern is that there may still be contention on main memory even if all other peripherals do not transmit on the bus. The CPU may concurrently attempt to access main memory. From a general framework perspective, we would need to pessimistically account for any increase in bus transfer time due to uncontrolled CPU interference through a technique similar to that used to analyze the reverse problem, CPU main memory delay because of peripheral interference (see Chapter 4). However, there is a key difference which makes the analysis easier, in that PCIe and PCI transactions are typically buffered within the interconnect. This means that unless the main memory bandwidth is exceeded, the peripheral will not notice any increase in transfer time because of the CPU accessing main memory (the reverse, however, is not true because the interconnect does not typically buffer CPU main memory access). In a typical COTS architecture, the cumulative bandwidth consumed by the CPU and each individual high-speed peripheral does not exceed the bandwidth of main memory, so calculating the I/O transfer time is straightforward.

The next concern is that, since individual bus transactions must be allowed to run to completion because of the COTS bus, the peripheral scheduler's `block` command is not acted upon immediately. In order to evaluate the impact of this delay, we computed its maximum value. A typical single lane PCIe device has an achievable bus throughput of 250MB/sec where each bus transaction is 4KB. This results in single bus transaction time of 16 microseconds. Since this is three orders of magnitude less than our I/O period (a high resolution video stream at 50 frames per second has an I/O period of 20ms), we consider its effect negligible for the schedulers we evaluate. However, if the peripheral scheduler uses a scheduler that switches between devices with the same granularity as the single bus transaction time (for example a least slack first scheduler), then this effect would need to be taken into account (for example by adding an `executing` output from each the real-time bridge to measure exact bus access time).

Prototype Details. In our prototype, the peripheral scheduler is built using the Xilinx ML505 Evaluation Platform [106] which features an XC5VLX50T FPGA. We created VHDL hardware code to implement the rate monotonic scheduling algorithm [58] for strictly periodic tasks, as well the sporadic server algorithm [93] to schedule aperiodic peripheral bus traffic. The implementation itself is split into two types of hardware components: 1) servers for each real-time bridge, and 2) a global scheduling algorithm. Each server generates a READY signal, and the global scheduling algorithm in this case executes the task with the highest static priority and an asserted READY signal.

In terms of FPGA area utilization, our implementation is lightweight. We synthesized a two layer peripheral scheduler with four sporadic servers scheduled according to a global RM scheduling policy. The resultant implementation

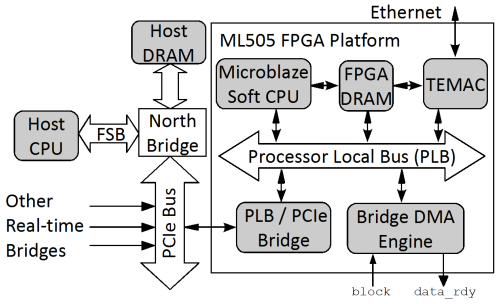


Figure 2.2: Real-Time Bridge Prototype, Block Diagram.

used 531 of the 28,800 available slice registers (1.8%), and 668 of the available 28,800 slice look up tables (LUTs) (2.3%). The number of I/O buffers used, which is associated with the chip's pinout, was also low (9 out of 480, 1.9%). These numbers lead us to conclude that the design will easily scale to concurrently manage dozens of I/O peripherals.

2.1.2 Real-Time Bridge

In order to provide real-time guarantees on bus communication, bus access must be controlled according to the policy dictated by the peripheral scheduler. Since off-the-shelf peripherals are unlikely to have such a mechanism built-in, a hardware control abstraction is interposed between each peripheral and the bus to provide this functionality to our real-time I/O management system. Each real-time bridge employs a buffer which is able to store pending traffic while bus access is prohibited. This allows high-bandwidth peripherals to be blocked from the bus for relatively long periods of time without suffering from data loss due to full internal buffers on the COTS peripheral. Combined with a communication guarantee provided by the peripheral scheduler's scheduling policy, this guarantees the I/O system will deliver all communication traffic by its I/O deadline.

We envision a final general real-time bridge using a setup similar to the ML455 [109], an FPGA development platform which can be directly interposed between a COTS bus and a COTS peripheral. This device contains both a PCI-X edge connector and a PCI-X socket slot connected to the same Virtex 4 FPGA chip, which would allow various types of peripherals to use the exact same real-time bridge. However, in order to rapidly develop a complete prototype, we focused our effort on a real-time bridge for one specific peripheral. We targeted a network interface card on the ML505 FPGA Evaluation Platform [106]. This device features both an Ethernet hardware interface as well as a one-lane PCIe edge connector both connected to the same Virtex 5 FPGA chip. We first describe the hardware components of the System-on-Chip (SoC) design, and then elaborate on the software development effort.

Hardware Design. A logical outline of the important hardware components in the network interface card version of the real-time bridge is shown in Figure 2.2. We now describe each of these in the order of the dataflow through the real-time bridge during normal operation. Consider the case where a packet arrives through the Ethernet connection.

First, the physical hardware interacts with the **Tri-state Ethernet MAC (TEMAC)** hardware block. This is a fixed hardware block on the FPGA, and is the COTS peripheral that the real-time bridge is managing. This block maintains a set of memory addresses where to place packets after they are received. After the packet arrives to the TEMAC, it gets stored into the **FPGA DRAM** and an interrupt is raised to the **Microblaze Soft CPU** [108] which provides information on where the packet was stored and the size of the received data. The Microblaze processor is a soft CPU, meaning that is implemented using the reconfigurable logic on the FPGA. We developed a driver running on the Microblaze that will take the addresses and lengths of the packets and put them into a download queue of data to be sent to the main system. This queue exists in two parts. The potentially long tail of the queue is stored in software, whereas a bounded number of entries (say 128) of the front of the queue are stored in hardware on the **Bridge DMA Engine**. The Bridge DMA Engine, which is a hardware block we created specifically for the real-time bridge, manages actually moving the data out of FPGA DRAM and transferring it into the host CPU's main memory. Along with the queue of data needing to be transferred, the Bridge DMA Engine manages the `block` and `data_rdy` signals on the real-time bridge. When `block` is asserted, the Bridge DMA Engine will not transfer data out of FPGA memory. The `data_rdy` signal is asserted whenever any data is in the hardware queue. The DMA transfers themselves are abstracted as an address to address copy on the Processor Local Bus (PLB). The **PLB / PCIe Bridge** handles the process of translating a write transaction on the PLB bus to a write transaction on the PCIe bus. When the Bridge DMA Engine is unblocked by the peripheral scheduler and performs a DMA operation, the memory containing the packet in FPGA DRAM is copied into the **Host DRAM**. After the transaction is complete, an interrupt is raised on the **Host CPU**, which then takes the packet data and passes it to the network stack for processing.

Sending packets out from the main CPU works in a similar way, but in the reverse order. The main CPU stores the packet data in Host DRAM and then writes the addresses to an upload queue which resides both in software as well as on the Bridge DMA Engine. When the Bridge DMA Engine is unblocked, it transfers the packets from Host RAM into FPGA DRAM (the PCI / PLB Bridge will again do address translation) and raises an interrupt to the Microblaze Soft Processor. Our driver on the Microblaze then sends the TEMAC hardware block the addresses and lengths of the packet data in FPGA DRAM. Finally, the TEMAC hardware sends the data over the physical Ethernet medium.

An important detail of this implementation is that the (comparatively) slow Microblaze processor does minimal processing (working with only the addresses and lengths) and no copying of the potentially high-bandwidth packet data. This allows our prototype implementation to achieve a network throughput of about 100 Mbps for upload and 80Mbps for download, which coincides exactly with Xilinx's TEMAC performance benchmarks for our setup (ML505, 125MHz Microblaze, 1500 Maximum Transmission Unit (MTU)) [38]. Performance can be further improved by using a larger MTU. Additionally, without the Microblaze bottleneck, the Bridge DMA Engine was able to send data at 207MBps, which approaches the theoretical limit of a PCIe single lane connection (250MBps).

It is worth noticing that the proposed control abstraction introduces additional latency compared with a COTS peripheral communicating directly to the PCIe bus. This is one tradeoff that is made in order to provide control

of peripheral bus access. We ran an experiment to get an idea of the effect of this additional latency by sending ping packets to the main CPU through our real-time bridge (scheduled as the highest-priority sporadic server) and sending ping packets directly to the FGPA's PetaLinux OS. Surprisingly, the packet round-trip times through our bus scheduling prototype (2.40ms) were actually *lower* than the round-trip times for the packets processed immediately on the FPGA (2.62ms). Hence, it is faster to place the packet data in the Bridge DMA Server, assert `data_rdy` to the peripheral scheduler, wait for the `block` signal to be deasserted, receive access to the PCIe bus, transmit the data into Host DRAM, process the ping on the host's 2.66 GHz CPU, and reverse the entire process for the ping response, than to handle the ping packet directly on the slower 125 MHz Microblaze processor. This experiment demonstrates the efficiency of our implementation.

This overall hardware framework currently supports a single real-time flow through each real-time bridge. In order to allow multiple real-time flows through a single real-time bridge, for example to distinguish different TCP connections on the same network interface, which vary in real-time importance, we would need to replicate the Bridge DMA Engine for each flow we would like to support. Since the scheduling wires interact only with the Bridge DMA Engine, the peripheral scheduler would not need to be modified. However, in such a design the FPGA Microblaze processor or a custom hardware block would need to read some packet data, for example the port, to distinguish between real-time flows and forward the address/length information to the corresponding Bridge DMA Engine, which may add overhead.

Software Design. The software architecture for the real-time bridge is paramount to our real-time I/O management system's applicability. Although the hardware design may be generalized for a particular bus interface and therefore reused, each unique COTS peripheral requires some software effort to be transparently controlled by a real-time bridge. In particular, each peripheral requires two drivers, a *host driver* to run on the main CPU, and an *FPGA driver* to run on the real-time bridge's Microblaze Soft CPU. A logical layout of their interactions is show in Figure 2.3.

One advantage of using COTS peripherals is that it is common for stable, open-source Linux drivers to already exist for peripherals that we are interested in controlling, which simplifies the software effort. Thus, in order to maximize code reuse, we run the Linux kernel on both the real-time bridge and the host CPU. The real-time bridge's FPGA runs PetaLinux version 0.30rc1[77], a Linux port for the Microblaze Soft CPU based on version 2.6.20 of the Linux kernel. The host system runs a Linux kernel, version 2.6.29. Importantly, the driver on the main system is designed to make the real-time bridge completely transparent to the end user and end-user applications. In Figure 2.3, the Host DMA Interface portion of the host driver and the FPGA DMA Interface portion of the FPGA driver can be reused in all real-time bridges. The high-level and low-level driver portions can be extracted from an open-source Linux driver for the targeted COTS peripheral. We now elaborate on each of the drivers.

The **host driver**, which is a kernel module running on the main CPU, creates a network card interface for the real-time bridge. From the user's perspective, there is no difference between using a network card directly and using a network card through a real-time bridge.

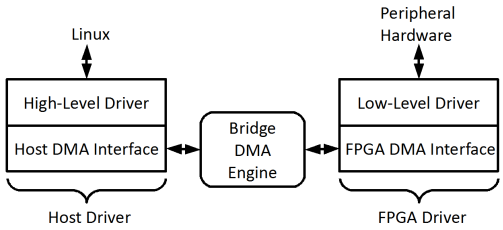


Figure 2.3: Software Architecture.

To handle incoming traffic from the network, the host driver allocates memory for incoming packets and maintains the *available addresses hardware queue* on the Bridge DMA Engine. When a packet arrives, the Bridge DMA Engine uses these addresses to store packet data and then may raise an interrupt. To reduce overhead, the Bridge DMA Engine does not raise an interrupt for every packet transferred. Instead an interrupt is raised when any of these conditions becomes true:

- one or more packets have been transferred, and the Bridge DMA Engine has no more packets ready to be transferred,
- one or more packets have been transferred, and the `block` line becomes asserted,
- the number of transferred packets since the last interrupt reaches a design-time parameter.

Every time an interrupt is raised, the driver, without making a copy, delivers the packet addresses and lengths to the Linux network layer. The Linux network layer then processes and frees the packets. Finally, the driver refills the Bridge DMA Engine's *available addresses hardware queue* with new addresses to replace those that were consumed.

For outgoing traffic, the driver receives packets from the Linux network layer and puts their lengths and addresses into the *outgoing packet hardware queue* on the Bridge DMA Engine. If the *outgoing packet hardware queue* fills up, the driver blocks the network layer until some packets have been transferred. The network layer, in turn, stores the backlogged packet information in a software queue. After the packet data is transferred to FPGA DRAM, an interrupt is raised by the Bridge DMA Server using the same rules as incoming packets, to inform the driver that the memory associated with the transferred packets can be freed.

The only dependence of the host driver on the COTS peripheral is the type of interface that is exported to Linux. This means that, if we were to design a real-time bridge for a different type of network interface card, the host driver would remain the same. To adapt the driver for a different class of peripherals, only the Linux interface would need be changed[2]. In terms of driver reuse, the Linux interface is contained in the high-level driver. Even if a Linux driver does not exist for the specific COTS peripheral we are using, all drivers for the same type of peripheral will contain

[2]The Linux interface determines the class of the device (character device, block device, or network interface), and the relevant device file operations.

the same Linux interface and therefore the same high-level driver. Thus, for developing the host driver, any existing driver for the same type of peripheral that we are using will reduce software effort.

The **FPGA Driver** runs on the Petalinux kernel on the real-time bridge. The driver consists of an FPGA DMA Interface, and the low-level driver for the TEMAC hardware block. The driver deals only with lengths and addresses of packets, and does not make copies or perform any processing in order to minimize the performance impact of the slow 125MHz Microblaze Processor.

Incoming and outgoing packets work as expected, with a few intricacies. When incoming packets become queued, perhaps because the `block` signal is asserted to the real-time bridge, the *incoming packet hardware queue* may become full. In this case, the driver maintains a software queue to store incoming packets until space becomes available in the *incoming packet hardware queue*. Therefore, packet drops and retransmissions are avoided resulting in better performance of network protocols like TCP. Additionally, the FPGA driver shares information with the Host Driver in order to propagate network parameters such as the MAC address and the Maximum Transmission Unit (MTU).

Even though the FPGA driver has direct interaction with the COTS hardware, most of the code can be reused from either the already-developed FPGA DMA interface, or the low-level driver portion of an existing Linux driver. For example, in our network card real-time bridge prototype, we use the existing TEMAC driver until the packets are ready to be given to the Linux network stack and then instead instruct the Bridge DMA Engine to send them to host memory. For outgoing packets, the data received from the Bridge DMA Engine is sent to the TEMAC hardware block using the same interface that the PetaLinux network layer uses.

In the worst case, if no driver source code is available for the COTS peripheral we want to control, the development effort required to develop the entire driver is still strictly less than the non-COTS approach: building application specific hardware in addition to the entire driver.

2.1.3 Formal Flow Analysis

The main advantage of the equivalence between I/O and CPU scheduling described in Section 2.1.1 is that feasibility can be checked using any suitable monoprocessor schedulability test, given either an I/O period and deadline information (for a periodic flow) or real-time server parameters (for an aperiodic flow). However, checking I/O feasibility is not enough to ensure system-level correctness. Data traffic must be processed by computational tasks, hence, in a distributed application, communication delay can significantly impact the overall end-to-end application delay. Furthermore, we must ensure that each real-time bridge has enough memory resources to buffer all the incoming data before it is transmitted on the bus, and similarly, that enough space is available in main memory to buffer all the outgoing data before it is fetched by the Bridge DMA Engine.

Delay and buffer space computation are easy for a periodic or sporadic flow. Techniques to compute worst case response time are available for both fixed priority [53] and EDF [94]. Let e be the maximum job size in bytes, p the period or interarrival time of the task, and r its response time. Then the maximum flow delay is simply r and the

required buffer size is $\lceil \frac{r}{p} \rceil e$. However, in the case of an aperiodic flow serviced by a budget-based server, the analysis is not so trivial. Note that incoming network flows in a distributed system are rarely exactly periodic: even if they are generated by periodic tasks and transmitted according to a highly predictable scheduling scheme such as TDMA, they likely accumulate jitter as they traverse multiple bridging and routing elements. Furthermore, clock skews are typically present due to lack of accurate timing synchronization in large distributed real-time systems.

To address this problem, we propose an analysis methodology for fixed priority scheduling based on the theory of Real-Time Calculus [98]. Real-Time Calculus extends the basic concepts of Network Calculus [16] and provides delay and buffer bounds for deterministic data and event flows. Network Calculus has been applied to model a variety of networking and routing scenarios, and is therefore well suited to provide characterization for incoming network data flows.

In Real-Time Calculus, a flow trace is described by a cumulative function $R(t)$, which represents the resource amount required by the flow in the interval $[0, t]$. A representation of all possible traces for the flow is provided by a tuple $\alpha(t) = \{\alpha^u(t), \alpha^l(t)\}$ of upper and lower arrival curves: $\alpha^u(t)$ is an upper bound on the resource requirement in any interval of length t, while $\alpha^l(t)$ is a lower bound on the resource requirement in the same interval. Formally:

$$\forall R, \forall s < t : \alpha^l(t-s) \leq R(t) - R(s) \leq \alpha^u(t-s). \tag{2.1}$$

Similar to arrival curves, a representation of the availability of a shared resource can be described by a tuple $\beta(t) = \{\beta^u(t), \beta^l(t)\}$ of upper and lower service curves: $\beta^u(t)$ represents an upper bound on the amount of service time provided to a backlogged flow during any interval of length t, while $\beta^l(t)$ represents the corresponding lower bound[3].

Physical resources are modeled by a collection of abstract components. Each abstract component receives an input flow $\alpha(t)$ and available resources $\beta(t)$, and produces an output flow $\alpha'(t)$ and service curve $\beta'(t)$ representing the remaining resources. The relations among input and output arrival and service curves depend on the processing semantics of the modeled flow and resource. For example, consider a flow that greedily consumes all offered resources when active. The corresponding abstract component can be described by the following relations [21]:

$$\alpha'^u = \min\{(\alpha^u \otimes \beta^u) \oslash \beta^l, \beta^u\}, \tag{2.2}$$

$$\alpha'^l = \min\{(\alpha^l \oslash \beta^u) \otimes \beta^l, \beta^l\}, \tag{2.3}$$

$$\beta'^u(t) = \inf_{\lambda \geq 0}\{\beta^u(t+\lambda) - \alpha^l(t+\lambda)\}, \tag{2.4}$$

$$\beta'^l(t) = \sup_{0 \leq \lambda \leq t}\{\beta^l(t-\lambda) - \alpha^u(t-\lambda)\}, \tag{2.5}$$

where the min-plus convolution $\otimes$ and the min-plus deconvolution $\oslash$ of two functions f and g are defined as:

$$(f \otimes g)(t) = \inf_{0 \leq \lambda \leq t}\{f(t-\lambda) + g(\lambda)\}, \tag{2.6}$$

$$(f \oslash g)(t) = \sup_{\lambda \geq t}\{f(t+\lambda) - g(\lambda)\}. \tag{2.7}$$

[3]Note that the concept of service curve in Real-Time Calculus is equivalent to that of strict service curve in Network Calculus.

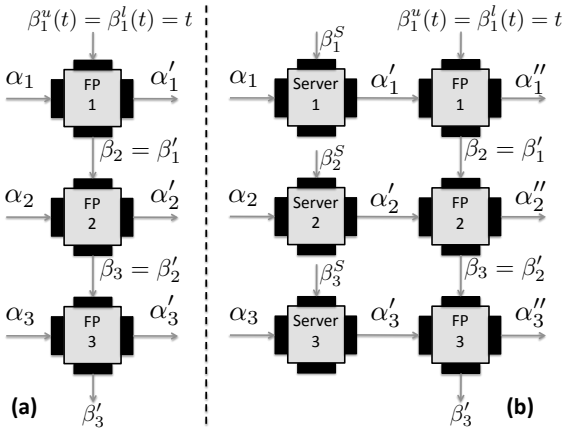


Figure 2.4: Abstract component interconnection.

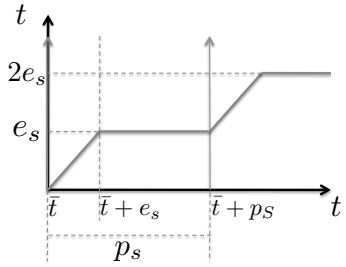


Figure 2.5: Service curve derivation, sporadic server.

Abstract components can be composed to model specific scheduling policies. For example, in Figure 2.4(a) we show how to model fixed priority arbitration for a system with three flows τ_1, τ_2, τ_3 in decreasing order of priority. The abstract component modeling flow τ_1 can use all available resources, the abstract component for τ_2 only gets the resources that were not consumed by τ_1, and so on. Finally, maximum delay $d^{\max}$ and required buffer space $b^{\max}$ for each flow can be computed based on input arrival and service curves as follows:

$$d^{\max} = \sup_{\lambda \geq 0}\{\inf\{t \geq 0 : \alpha^u(\lambda) \leq \beta^l(\lambda + t)\}\}, \tag{2.8}$$

$$b^{\max} = \sup_{\lambda \geq 0}\{\alpha^u(\lambda) - \beta^l(\lambda)\}. \tag{2.9}$$

To account for the effect of scheduling servers, we propose to add an additional abstract component for each flow as shown in Figure 2.4(b). In our model, each flow is characterized by three arrival curve tuples: α, α' and α''. For received network traffic, α represents the input flow to the real-time bridge, and α'' represents the flow transmitted on the PCI bus. Resource requirements are expressed in terms of transmission time by the real-time bridge. Let C be the achievable transmission speed of the real-time bridge. Then, if a maximum of e bytes can arrive in an interval of length $\bar{t}$, it follows that $\alpha^u(\bar{t}) = \frac{e}{C}$. As a simple example, a strictly periodic input flow with period p and constant job size of e bytes can be modeled with arrival curves $a^u(t) = \lceil \frac{t}{p} \rceil \frac{e}{C}, a^l(t) = \lfloor \frac{t}{p} \rfloor \frac{e}{C}$. Finally, α' is the virtual output flow generated by the abstract server component based on the scheduling server processing semantics. In each interval of length t, $\alpha'^u(t), \alpha'^l(t)$ are upper and lower bounds on the transmission time required by the server. In general, for a budget-based server with budget e_s and period p_s, α' is lower than α, since the server constrains the task to require a transmission time of no more than e_s time units every period p_s.

Next, we determine the way in which to compute service curves β^S for an abstract server component. An end-to-end service curve for each flow can be obtained by concatenating β^{Sl} and β^l computing $\beta^{Sl} \otimes \beta^l$ [16], and delay and buffer bounds can be obtained using the concatenated service curve in Equations 2.8 and 2.9.

Theorem 3 *A feasibly scheduled sporadic server with budget e_s and period p_s can be modeled by an abstract component with service curves:*

$$\beta^{Su}(t) = \min\left(\left\lceil \frac{t}{p_s} \right\rceil e_s, t - \left\lfloor \frac{t}{p_s} \right\rfloor (p_s - e_s)\right) \tag{2.10}$$

$$\beta^{Sl}(t) = \begin{cases} \beta^{Su}\big(t - (p_s - e_s)\big) & \text{if } t \geq p_s - e_s \\ 0 & \text{otherwise} \end{cases} \tag{2.11}$$

Proof.

Consider an interval $[\bar{t}, \bar{t} + t]$ during which the task serviced by the sporadic server is constantly backlogged. Then, an upper bound $\beta^{Su}(t)$ for the service provided by the server to the task in interval $[\bar{t}, \bar{t} + t]$, which is equivalent to the transmission time requested by the server to the fixed-priority scheduler in the same interval, can be obtained assuming that the budget is recharged to the maximum e_s at time $\bar{t}$. This results in the server providing e_s units of service time from $\bar{t}$ to $\bar{t} + e_s$ and again in each interval $[\bar{t} + kp_s, \bar{t} + kp_s + e_s], k \geq 0$. The obtained service curve, shown in Figure 2.1.3, is captured by Equation 2.10.

Now consider the lower service curve $\beta^{Sl}(t)$. Since the task is backlogged and the system is assumed schedulable, the server must provide e_s units of service time to the task every p_s time units. A lower service curve can thus be obtained assuming that the server has no remaining budget for the longest possible interval $p_s - e_s$ before fully recharging it at time $\bar{t} + (p_s - e_s)$. Afterwards the curve is equivalent to the upper service curve, and we can compute it as $\beta^{Sl}(t) = \beta^{Su}\big(t - (p_s - e_s)\big)$. □

While we derived β^S in the case of a sporadic server, the analysis easily extends to other server types by deriving suitable best and worst case arrival patterns.

2.1.4 Evaluation

The goal of our evaluation is twofold. First, we demonstrate that there is a problem using COTS interconnect for a real-time system. We present an I/O task set which results in I/O deadline misses when running on a standard COTS bus. Next, we run the same I/O task set within our scheduling framework, and show that all deadlines are met. This demonstrates the non-real-time nature of COTS interconnects, and validates the correctness of our solution.

Performing direct measurements on a high performance COTS I/O system such as PCIe is difficult: the PCIe protocol implements point-to-point connections between each peripheral and the rest of the system running at the very high speed of 2.5 Ghz, making it hard to directly observe. In order to make the most accurate measurements, we used dedicated hardware on the reservation controller. Our trace acquisition hardware module polls the state of the `data_rdy` and `block` signals with a one microsecond resolution. Any changes in these signals, along with an

Board	Data Size	Transfer Time	Budget	Period
ML555	4.0 MB	4.4 ms	5 ms	8 ms
ML505	1.1 MB	7.5 ms	9 ms	72 ms
ML505	1.1 MB	7.5 ms	9 ms	72 ms
ML505	1.1 MB	7.5 ms	9 ms	72 ms

Table 2.1: Flow parameters.

associated timestamp, are output over the reservation controller ML505's serial port where they can be received by an external computer for processing.

We performed experiments on a COTS PC platform with an Intel 975X system controller (northbridge). The selected motherboard has four PCIe slots, allowing us to connect up to four high-speed peripherals. Using a PC platform permits easy access to all PCI slots, however, to derive meaningful measurements, we changed the FSB clock frequency obtaining a theoretical memory bandwidth of 2.4Gbyte/s, which is in line with typical values for embedded platforms.

To make our experiments more easily repeatable, we instructed the real-time bridge prototype to generate synthetic traffic instead of using traffic received by the TEMAC over the network. Our periodic task generating drivers run on the main CPU, and since our I/O schedule uses periods on the order of milliseconds, it is difficult to exactly synchronize all synthetic tasks. For this reason, we ran the tests for many hyperperiods, and show here the traces from the most closely aligned arrival times, which correspond to the *near-critical instants*. The arrival times of the presented traces are never separated by more than 0.8 milliseconds. Additionally, we implemented a traffic generator using an ML555 PCI Express Development Board [107] with a faster 8 lane PCIe connection. The synthetic traffic generator is programmed to send a constant amount of data to main memory every period and obeys the I/O scheduling commands from the reservation controller.

The task set used in our experiments consists of four real-time flows competing for main memory. The task parameters (data size, transfer time, period) are shown in Table 2.1. The tasks' periods are harmonic, and the total utilization does not exceed 100%, so the task set is schedulable under RM.

In the first experiment, the COTS bus is used without the reservation controller. Traffic gets sent on the bus as soon as it arrives, increasing the execution time of the ML555's periodic task from 4.4 ms to over 8 ms (an increase of 82%) when the tasks start at a near-critical instant. This causes a deadline miss (see Figure 2.6). In the second experiment, each peripheral is handled by a sporadic server (whose corresponding budget and period are shown in Table 2.1) and all the servers are scheduled according to Rate Monotonic with total utilization $\frac{5}{8} + \frac{9}{72} + \frac{9}{72} + \frac{9}{72} = 1$. By using the proposed real-time I/O management system, the task set is now successfully scheduled without missing deadlines because the traffic is prioritized. A trace of one hyperperiod starting at a near-critical instant is shown in Figure 2.7.

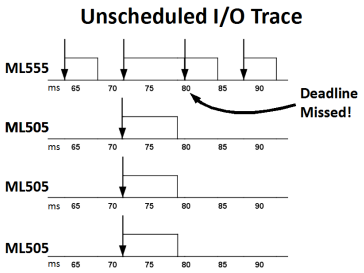


Figure 2.6: Trace with no peripheral scheduling.

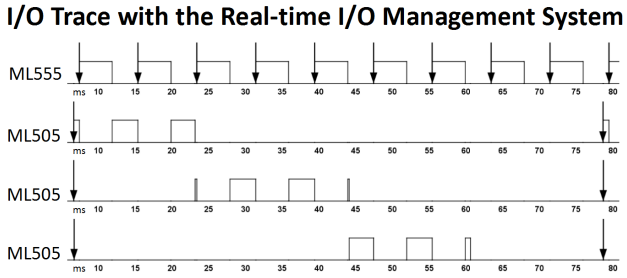


Figure 2.7: Trace with peripheral scheduling.

2.1.5 Related Work

To the best of our knowledge, this is the first work that attempts to predictability schedule peripheral traffic in a COTS PC system. However, there exists related work that deals with modeling and analysis of similar architectures. Modeling complex COTS interconnections and estimating delay and buffer requirements for peripheral flows can be done in an AADL-based environment [65]. An event-based model may be used to estimate delay for both computation and communication activities in a multicore system-on-chip [85]. However, lack of precise knowledge of COTS behavior implies that these analyses must make pessimistic assumptions, which can lead to high delay and buffer sizes. Our real-time I/O management system removes such unpredictability by forcing an implicit bus schedule.

Several other analysis frameworks exist to estimate delay characteristics for communication flows in embedded systems. For example, Real-Time Calculus can compute end-to-end delay for various real-time systems [98, 102]. Analysis methodologies are available for existing real-time interconnections such as CAN [99]. However, these methodologies typically assume a detailed knowledge of each component's behavior, which is unavailable in a COTS-based system.

2.2 SoC Platform for Mixed-Criticality Systems

Systems-on-Chip (SoCs) are increasingly popular in the embedded system domain because they consume less power and cost less money than the multi-chip solutions they replace. We believe that the SoC paradigm is especially promising for safety-critical embedded systems such as those employed in the medical market [103]: it is easier to formally verify the correctness of critical functionalities when they are implemented in hardware rather than in software on top of an OS and library stack [10]. Furthermore, SoC design can offer higher hardware reliability.

In this section, we describe new hardware control abstractions for SoC processing elements that provide strong physical isolation guarantees and impose limits to fault propagation. To automatize the insertion of hardware abstractions as much as possible and reduce the possibility of human errors, our control mechanisms are introduced together with a new design methodology for SoC that specifically targets mixed-criticality systems.

Our methodology is similar to the concept of *Platform-Based Design* (PBD) [84]. A platform is a library of (usu-

ally parametric) components. A platform instance is a set of library components selected to generate a concrete design. In a PBD design flow, the designer first specifies the system functionality using an implementation-independent description language. The designer then selects a suitable platform and performs *mapping* of functional elements to platform components, thus creating a platform instance. Our platform contains three types of architectural components: processors (which can either be CPUs or *hardware processors*, e.g. logic circuits implementing a specific functionality), communication infrastructures and memories. Functional specification, architectural specification and mapping are all performed using the Architectural Analysis and Design Language (AADL) [34]. AADL is rapidly gaining support in safety-critical markets such as avionic and medical domain and it has been applied to many industrial examples (see Section 2.2.1). Mapping is performed by binding functional processes to architectural processors and specifying processes' requirements in term of data transfers (functional isolation) and resource usage (physical isolation). This requirement specification creates a *certificate* for each process, e.g. it determines the complete set of acceptable run-time behaviors for the process.

Isolation is provided by encapsulating each processor in an hardware abstraction called a *monitoring hardware wrapper*. The wrapper controls all communication and resource usage by the processes executed on that processor, and can therefore enforce their certificates at run-time. Note that the platform can not prevent a process from failing if it suffers a critical internal error. As such, the certificate for a low-criticality, unverified process must include a halting behavior, while safety-critical processes must be formally verified/certified to prevent such internal errors. The main goal of the platform is to simplify system level verification: to obtain a safe system, it is sufficient to verify that all safety-critical processes are correct assuming that low-criticality processes behave according to their certificates. Without certificate enforcement, a low-criticality process could exhibit any kind of byzantine failure, making verification either extremely hard or entirely impossible.

The rest of the section is organized as follows. Section 2.2.1 discusses related work. Our AADL-based system modeling is introduced in Section 2.2.2, together with a case study based on a medical pacemaker. Section 2.2.4 describes our architectural implementation, focusing on the wrappers. Finally, Section 2.2.5 provides more details on how the wrappers can be used to enforce timing isolation; run-time monitoring for functional isolation is described in Chapter 3.

2.2.1 Related Work

The concept of PBD is well established and several SoC platforms are commercially available [24, 66]; however, they are not targeted at safety-critical systems. Similarly, a variety of PBD and model-driven design methodologies have been proposed (see [84] for an overview), several of which (for example, see [30, 48]) support formal verification of modeled application behavior. The main difference is that our methodology and platform does not simply allow to verify the correctness of a high-level model: it enforces application isolation at run-time, as long as the designer can correctly specify system isolation requirements in the model.

AADL-based tools and analysis methodologies have been developed in the area of safety analysis [96], dependability [83] and schedulability [92, 45, 91]; however, to the best of our knowledge this is the first time that an AADL-based model is used to automatically generate code that enforces safe process behavior.

There have been designs to ensure the safety of pacemaker systems in terms of the control algorithm [10], including model-checking of the pacing controller using UPPAAL [101]. However, such previous work is based on the assumption that the underlying implementation can offer strong isolation guarantees for memory, power and communication. In this work, we take a more general approach where isolation guarantees are not a by-product of the system implementation, but are instead specified as requirements in the functional model.

2.2.2 AADL-Based Modeling

The importance of model-based Architecture Description Languages (ADLs) and formal analysis of system models is becoming apparent in the industry of hard real-time systems, such as avionics, aerospace, and medical devices. Model-based ADLs provide the automation capability of generating beneficial abstractions from system models to be verified or examined by computed-aided tools. AADL is a type of ADL initially developed for the avionic market. It is based on 15 years of research, including the MetaH language developed by Honeywell Labs and several DARPA programs [15]. In this section, we first describe how AADL is used in functional specification and mapping for our platform. Then, in Section 2.2.3 we introduce our case study of a medical pacemaker.

AADL lets designers specify the logical functional model separately from the hardware platform. An AADL specification is comprised of different components and their interactions. Components used for the logical design include `process`, `thread`, and `data`. AADL execution platform components include `processor`, `memory` and `bus`; they have a one-to-one correspondence with our architectural components. Each `processor` represents either a custom-built hardware processor or a CPU. `Memory` is used, among others, for all external memories (such as external SRAM or DRAM chips) used to hold shared data. `Bus` represents the unique communication infrastructure used to interconnect all processors and memories. All components are tagged with properties which add extra information that can not be expressed by structural descriptions (for example, the processor type is specified by a `class` property). The core AADL standard has pre-declared properties that support real-time scheduling as well as other areas of research. AADL also provides the syntax to add new user-defined properties.

Every application is modeled as a collection of `process`es with one or more `thread` subcomponents. Threads are active agents: they receive inputs, process and output data. Period, deadline, and execution time properties are associated with each `process` as a timing reservation for all of its subcomponent threads. The platform guarantees that at each period a process is provided with an amount of execution time at least equal to its request within a time window equal to the specified deadline. Each `process` represents a dedicated memory and design space protected by the system architecture. The platform ensures that all threads inside a `process` are functionally isolated from other threads running on different `process`es.

Two types of interprocess communication are supported. A process can declare any number of input *message queues* to which other processes can send data. Furthermore, shared data objects can be declared and accessed by any process. In AADL, message queues are modeled as `event data port connection`s and `data access`es are used to connect processes to shared data object. Each communication has an associated `data` component which represents the type and size of data that is to be sent or shared and an associated `deadline` property which represents the maximum allowed amount of time to complete the data transfer. The acceptable communication behavior of a process can be further specified by a `PropertyList` and associated `EventList`s. Each entry in an `EventList` describes a specific communication event, e.g. a send/receive to a message queue or a DMA read/write to a data object. Properties specify behavior as a formal trace of events at run-time, described by either a formula in Past-Time Linear Temporal Logic (PTLTL) or an Extended Regular Expression (ERE) pattern (see Chapter 3).

Platform mapping is performed by ”binding” logical components to architectural components. Each `process` is bound to a `processor` and each `data` component is bound to a `memory` (either a message queue or an external memory). Each HW processor can execute a single process, while a CPU processor can execute multiple processes.

The AADL platform specification would not be very useful if the designer had to manually check it for correctness and convert it into an implementation, since the potential for human error during translation is very high. As such, we built a set of tools to assist the design flow. Our AADL tool makes sure that the model conforms to predefined *analysis specific* rules for correctness; these include the aforementioned binding rules. Also note that some properties of logical components depend on the binding, and their values must be re-examined every time a related change in the mapping occurs. Some property values are derived automatically by the tool (in particular, addresses are associated with all data and memory elements), while others must be specified by the designer. For example, required execution time depends on the processor to which a process is bound, and can not be automatically computed without detailed code knowledge. Similarly, message queue size depends on the process behavior. The details of relevant properties and how they are computed will be introduced in subsequent Sections 2.2.4, 2.2.5. After binding is complete, the tool generates an XML file based on a customized schema containing a description of the platform instance; the XML file is then read by synthesis tools that generate the wrappers (see Section 2.2.4). In particular, each process is characterized by a certificate specifying functional behavior and timing requirements. The *functional certificate* is composed of the `PropertyList` and associated `EventList`s; property and event syntax is described in Chapter 3. The *timing certificate* is based on the computation and communication requirements of the tasks and expresses a set of timing reservation; its derivation is detailed in Section 2.2.5.

2.2.3 Case Study: Medical Pacemaker

Pacemakers are one of the most critical medical devices [103]. Once they are implanted, they must continue to operate correctly with very little maintenance for at least 5 years. A typical functional decomposition of cardiac pacemaker is shown in Figure 2.8. A pacemaker is connected to the heart via leads (typically two). These leads provide EKG

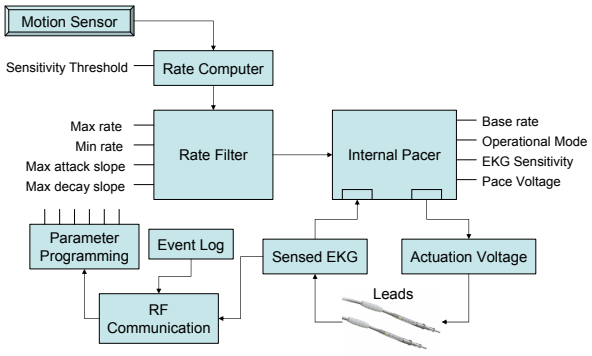


Figure 2.8: Pacemaker Block Diagram

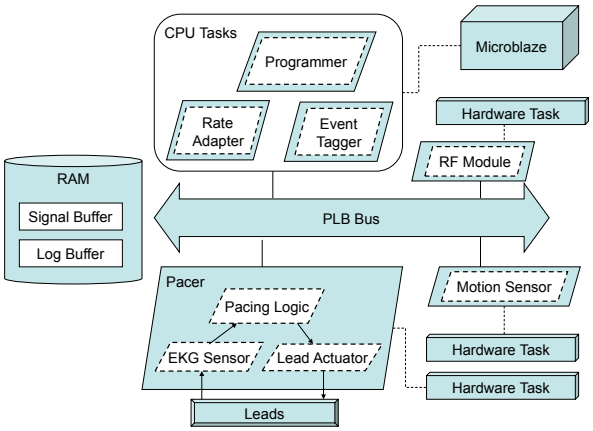


Figure 2.9: Mapped Pacemaker Platform

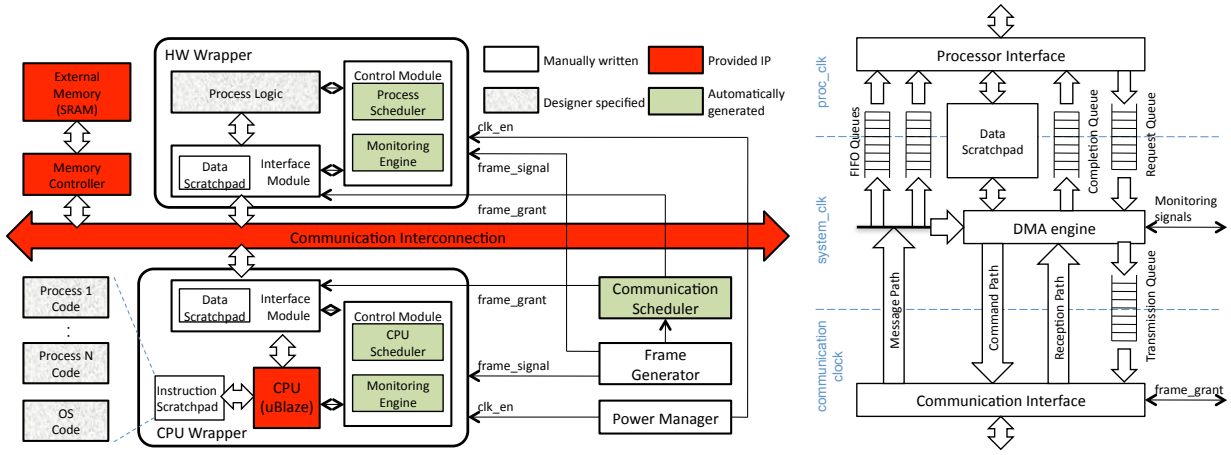
signals which the internal pacer uses to detect whether the heart is contracting properly and to send shocks to the heart to force contraction. The internal pacer has various control parameters which can be tuned to each patient: these include a reference rate of pacing, the operational mode, and the thresholds for sensing and actuation. The operational mode determines how to pace the heart; modes can determine which leads are used for sensing if any, whether rate adaptation is used, and whether certain types of special therapies should be provided.

A simple pacemaker only needs the leads and the internal pacer to be functional. However, for more effective pacing, it is desired to have the pacing rate adapt to the activity levels of the patient. Rate adaptation normally uses some form of motion sensing, usually through accelerometers or pressure sensors. The motion data is then used to compute a rate. A rate computation uses a sensitivity threshold to filter out motion noise (i.e. when riding a car). Furthermore, to match biological characteristics, the computed heart rates values are bounded (min and max rate) as well as their rates of change (attack and decay slope).

Finally, the pacemaker employs a communication component using RF signals. RF communication allows medical personnel to program various parameters of the pacemaker without intrusively removing the pacemaker. Furthermore, logged events and real time sensor data can be sent to the medical personnel during diagnostics and check-up.

Based on this discussion, we define three criticality levels in our pacemaker design: a *life-critical* level for the wires and internal pacer; a *mission-critical* level for the rate adaptation, whose failure is not life-critical but could still cause significant discomfort to the patient; and a *non-critical* level for communication and logging.

A schematic of an AADL-modeled pacemaker platform instance is shown in Figure 2.9. The core pacing is implemented by the life-critical *Pacer* process in a HW processor. The Pacer includes the sensor and actuation interfaces to the leads and also the pacing logic. Life-critical *EKG Sensor* data arrives at the frequency of 128 16-bit samples per second, which also determines the Pacer period. Also, the Pacer logs *EKG Signal* values and *Shock Event*s in the *signalBuffer* memory area for later retrieval and diagnosis. The values being written to the signalBuffer are timestamped and will eventually be overwritten in a circular manner. The mission-critical *Motion Sensor* process is also



(a) ***Architecture Block Diagram*** **(b)** ***Interface Module Block Diagram***

Figure 2.10: Pacemaker Platform Instance

implemented in hardware. It samples data at 80Hz and sends the measured values to the *Rate Adapter* through a message queue.

Three processes requiring complex functionality are executed on a Microblaze CPU. The mission-critical Rate Adapter uses motion data to compute a reasonable rate required for pacing and sends it to the Pacer twice per second. The non-critical *Event Tagger*, running at 8Hz, reads in a window of EKG signal values and shock events from the signalBuffer and finds anomalies that are then logged to another section of memory (the *logBuffer*) for more permanent storage and later diagnostics. Finally the non-critical *Programmer* is used to send parameter updates. It processes commands sent by medical personal through the non-critical *RF Module* (implemented in hardware) and it sends rate adaptation parameters (ie. max/min rate, attack/decay slope) to the Rate Adapter process and pacing parameters (ie. sensor/actuator thresholds, base rate) to the Pacer process through message queues.

2.2.4 Architecture Implementation

In this section, we describe the architectural components that comprise our platform focusing on how the monitoring wrappers can enforce isolation. Our general platform principles are orthogonal to the specific chip fabrication process being used, albeit certain implementation details and the choice of available IPs necessarily depend on it. Our pacemaker prototype implementation is based on a Xilinx Virtex-5 FPGA [111].

Figure 2.10(a) shows a block diagram for an example platform instance. For simplicity, the example is composed of only one hardware processor, one CPU and one external memory, but note that in our prototype implementation the number of CPUs, HW processors and memories is only limited by the available chip resources. Monitoring wrappers depend on the type of processor they control: *HW wrappers* and *CPU wrappers* are employed for hardware processors and CPU respectively. Each wrapper is comprised of two modules: an *interface module* and a *control*

module. The interface module mediates access from the processor to the communication infrastructure and provides standardized communication services. Processes can only exchange information by transmitting data through their interface module[4]. This effectively prevents fault propagation since the interface module is able to reject any data transfer that would lead to a certificate violation. The control module implements the actual monitoring logic: it checks all data transfers performed by the interface module against the property specification included in the certificate, and enforces the process timing reservation. Similarly, each external memory is connected to the communication infrastructure by a memory controller; however, since memories are passive component, no monitoring is required. Finally, the platform includes three additional *global modules*: the *Communication Scheduler*, the *Frame Generator*, and the *Power Manager*, which are used to enforce communication timing reservation and power consumption.

We can distinguish among four types of blocks in Figure 2.10(a): 1) blocks that are generated by the designer. These include the hardware logic of each HW processor implementing the executed process and the code for each software process executed on a CPU. 2) Blocks that are automatically generated by our implementation tools. These include the portion of the control module that implements the certificate and the Communication Scheduler. 3) Blocks that represent available IPs. These include: A) memories and their controllers; B) the communication infrastructure; C) CPU and associated Operating System (OS). 4) Blocks that are manually written by the platform provider, and depend both on the fabrication process and the supported IPs. These include the wrappers, Frame Generator and Power Manager. We implemented these blocks in a parameterized, mixed VHDL/Verilog Register-Transfer-Level (RTL) description.

The platform can support a variety of different IPs, albeit due to time constraints we only implemented wrapper support for some of them. However, specific constraints must be imposed on IPs to ensure required isolation and timing predictability. In what follows, we specify the required constraints for A) memories, B) communication infrastructure and C) CPU and OS. A) The access time to external memories should be predictable, or at least upper bounds must be easily computable. Our pacemaker prototype employs an external SRAM, which offers deterministic access time. Predictable DRAM controllers for real-time systems have been described in the literature [3]. B) Shared buses, segmented buses and Networks-on-Chips (NoCs) can all be employed as communication infrastructure, but the specific IP choice impacts the derivation of the communication schedule as described in Section 2.2.5. For simplicity, in the rest of this section we consider a simple shared bus, deferring the analysis of more complex infrastructures to future work; our prototype employs the IBM PLB [46]. C) We do not allow caches on CPU. While caches can greatly improve average computation times for general purpose computing, their inherent unpredictability makes it very hard to obtain tight computation time bounds for processes and to schedule cache miss traffic. Instead, we provide each CPU with local instruction and data memory used as *scratchpads*. Scratchpads can be as fast as caches, but data transfers to/from external memory must be explicitly initiated by the running process instead of being handled by the cache controller. While this imposes additional responsibility to the designer, advantages in term of timing predictability

[4]Note that since processors can implement external I/O, processes could potentially communicate through the environment. We assume that designers do not intentionally introduce such covert channels, and any environment dependency is solved through application of control theory.

and simplicity of implementation are well documented both in academia [56] and industry [23]. Furthermore, a large number of embedded CPU available either as soft or firm IPs for SoC design have configurable memory paths, and can therefore be easily connected to custom-designed scratchpads. Our platform can implement separate instruction and data scratchpads to support Harvard CPU architectures. In particular, our prototype supports the Xilinx Microblaze soft CPU running the Xilkernel OS. We shall assume that the code of all processes and of the OS executed on the CPU fits in the instruction scratchpad, which is typically possible thanks to the small footprint of embedded processes. The data scratchpad is controlled by the interface module: the processor can request it to transfer data from/to any external memory to/from the data scrachpad. The designer specifies the size of both the instruction and data scratchpad during mapping. Finally, if multiple processes are executed on the same processor, additional isolation requirements are imposed on the CPU and OS. In particular, the CPU must provide memory protection since scratchpads are shared. Furthermore, the OS must provide strong code isolation, especially regarding shared memory structures and libraries. A full description of OS-level isolation techniques is outside the scope of this work. As an example, the ARINC653 avionic standard [2] prescribes a set of requirements for the safe integration of mixed-criticality applications on a shared CPU, and ARINC653 certified OSs are available on the market.

An important note is relative to the criticality of the various platform blocks. Any fault in the communication infrastructure, external memories and controllers, global modules or in any monitoring wrapper can potentially compromise the whole system. As such, these components must be formally verified and/or certified to the highest degree of safety required by any application executed on the platform. While this can be expensive for the communication infrastructure and memories, we argue that if any form of communication and memory sharing is required in a mixed-criticality system, such requirement can not be avoided. The monitoring wrappers and global modules, in particular the control module and all schedulers, has been designed to be relatively easy to certify. A formal proof of the correctness of generated run-time monitors is available [22], albeit this does not remove the need for certification of the final implemented wrappers, since it is unfeasible to formally verify FPGA implementation tools like the placer and router.

Global Modules

Three global modules, the Frame Generator, Communication Scheduler and Power Manager, are included in each platform instance to manage system-wide behavior. Each HW component in our platform can be clocked independently. In particular, each monitoring wrapper employs two distinct clocks: a `system_clk` that is used for the control module and interface module, and has the same frequency for all wrappers; and a `proc_clk` that is used for the processor. In this way, different CPU and HW processors can run at different frequencies. As the integration scale of SoC becomes larger and clock frequency increases, it becomes impossible to synchronize all HW modules based on the same clock because signal propagation delay grows larger compared to the clock period. As such, large ASIC designs are moving towards a Globally-Asynchronous, Locally-Synchronous (GALS) paradigm where individual modules are based on synchronous logic but inter-module communication is asynchronous. Unfortunately, asynchronous communication is

ill-suited to safety-critical systems: formally verifying asynchronous systems is order of magnitudes more complex when compared to synchronous design.

To solve this problem, we divide time into fixed-length intervals called *frames*. The Frame Generator periodically produces a `frame_signal` that is propagated to all wrappers; the frame period is significantly larger than both the physical clocks (in our prototype, we use a frame period of 1us, while the `system_clk` has a 8ns period) and signal propagation delays. All timing requirements (period, deadline and execution time) expressed in the AADL model are multiples of the frame period, and processors are synchronized based on frames. In particular, processes are periodically activated and communication is initiated on a frame boundary.

The Communication Scheduler uses the frame information to control data transmission on the communication infrastructure. For each process, a `frame_grant` signal is propagated to the corresponding monitoring wrapper: during each frame, the interface module is allowed to transmit on behalf of the process only if the provided `frame_grant` signal is set. Internally, the Communication Scheduler implements a scheduling table based on a fixed length hyperperiod: for each frame in the hyperperiod, the table determines which processes are allowed to transmit. In practice, since a same set of processes could be allowed to transmit for an interval of several frames in a row, the table encodes the length of every such interval. To ensure timing predictability, the table is built to enforce *contentionless communication*: two processes can be allowed to transmit in the same frame only if their data transmissions can be carried out in parallel. In particular, since our prototype employs a shared bus, we allow a single process to transmit in each frame; if the communication infrastructure were implemented as a segmented bus connected by bridges, processes using different bus segments could be allowed to transmit simultaneously. Apart from timing predictability, the contentionless principle can simplify infrastructure design: for example, wormhole routers with no buffers can be used in a NoCs [39].

Finally, the Power Manager monitors power consumption. A `clk_en` signal is propagated to each monitoring wrapper. The `proc_clk` is periodically generated only while the `clk_en` is high; therefore, the Power Manager can completely stop a processor by simply lowering the corresponding `clk_en` signal. In our implemented prototype, the Power Manager periodically checks the input voltage to the system using the System Monitor functionality of the Virtex-5 FPGA; in a battery-operated system such as a pacemaker, this can be used to derive an estimate of remaining battery energy. If the energy becomes dangerously low, the Power Manager can shut down the non-life critical systems running on the CPU to save energy for the base pacing module. If additional measurement functionalities were available on the chip, the Power Manager could implement more refined control actions. For example, if power consumption could be measured for each processor, a misbehaving processor consuming an excessive amount of energy could be selectively turned off.

Interface and Control Module

Figure 2.10(b) shows a detailed block diagram of the interface module for a HW processor: it is composed of three main submodules, the *Processor Interface*, the *DMA Engine*, and the *Communication Interface*. The Processor Interface is directly connected to the processor and exports the communication services; it uses the same `proc_clk` as the processor. The DMA Engine uses the `system_clk` and implements most of the interface module logic. Finally, the Communication Interface connects to the communication infrastructure and shares its physical clock: it converts data transfer commands issued by the DMA Engine into read/write transactions on the communication infrastructure. This division of concerns simplifies development and certification: a new Communication Interface must be implemented for each communication infrastructure IP, but the DMA Engine is fully reusable. The Communication Interface receives the `frame_grant` as input, and it is allowed to read/write only when the signal is high. In practice, to account for the effect of propagation delay, after `frame_grant` goes high the Communication Interface must wait for a guard time equal to twice the maximum signal propagation delay before it can start to transmit.

Since all three submodules lie in different clock regions, dual-port memory elements are used to connect them. The *data scratchpad* can be simultaneously accessed by both the Processor Interface and DMA Engine. A variable number of *FIFO queues* are used to hold incoming data to message queues; the processor can read from the FIFO queues at any time. Service request by the processor are sent to the *request queue*. There are two types of requests: 1) sending a message to the message queue of another processor; 2) performing either a read (from external memory to data scratchpad) or write (viceversa) DMA operation. A message transfer request contains an ID for the destination queue and the data to be sent, while a DMA request contains the length of the transfer and the base addresses in both external memory and the scratchpad. All transfers are multiple of a 32-bits word. The *completion queue* is used to signal the end of a DMA operation back to the processor. Finally, the *transfer queue* is used to connect the DMA Engine to the Communication Interface.

The DMA Engine processes incoming service requests in order based on the request type. First of all, for every message transfer it translates the queue ID into an address. All external memories and message queues are automatically assigned unique system-wide addresses by our tools. The address is used by the communication infrastructure and Communication Interface to determine the destination of a data transfer. Second, for message transfers and DMA write, the DMA Engine simultaneously moves data from the request queue or the data scratchpad to the transfer queue, and sends the data to the *Run-Time Monitoring Engine* in the control module. The engine is responsible for checking each transferred data word against the functional properties included in the certificate, and it issues either a reject or accept command in 4 clock cycles. If the transfer is rejected, the transfer queue is flushed. Otherwise, the DMA Engine commands the Communication Interface to start the data transfer. The DMA Engine then immediately proceeds to service the next request; in particular, to avoid stalling the Communication Interface, it can write to the transfer queue while a previously accepted data transfer is being carried out. Incoming data from DMA read operations is forwarded to the Run-Time Monitoring Engine but never rejected, hence a reception queue is not required; this is allowed because

Module Name	Area (slices)
Virtex-5 VLX50T FPGA	28800
Communication Scheduler + Frame Generator	92
Interface Module (pacer, single task)	1830
CPU Scheduler (excluding CPU port)	68
Process Scheduler	38
Monitoring Engine (Event Tagger)	77
Monitoring Engine (Programmer)	202

Table 2.2: FPGA Resource Usage

read operations can not cause side-effects in the system. Similarly, incoming messages are monitored but immediately injected into the corresponding message queue.

The control module for a HW processor is composed of two main submodules. The aforementioned run-time monitoring engine is described in more details in Chapter 3. The process scheduler receives the `frame_signal` as input and periodically activates the HW process through a `process_start` wire. The process is responsible for signaling the end of its periodic execution through a `process_stop` signal. Failure to do so denotes a critical error; as a consequence, the process is stopped by halting its `proc_clk` clock.

The interface and control module for a CPU processor have some added complexity, mainly because multiple processes can be executed on the same CPU. Message queues, request and completion queues, transfer queues and Run-Time Monitoring Engines are duplicated for each process; our wrapper VHDL description is parametric in the number of processes and our Microblaze implementation supports up to ten. Note that we expect few processes to be mapped on each CPU (typically one per executed application), since each process can have multiple threads. One `frame_grant` signal for each executed process is provided to both the DMA Engine and Communication Interface; they each service the process for which the `frame_grant` is high. The Processor Interface is customized based on the CPU IP; in our implementation for the Microblaze all services are exported through memory-addressable registers. Finally, the process scheduler is replaced with a CPU scheduler. It implements a scheduling table for processes which is similar to the table in the Communication Scheduler. Whenever a different process must be executed, the CPU scheduler interrupts the CPU and communicates the ID of the process. In this way we make sure that computation is synchronized at the frame boundary, while requiring only minimal kernel modification. Note that while we do not support it in our Xilkernel implementation, the OS can implement a two-level scheduler (as required for example by ARINC653), where the time assigned to each process by the CPU Scheduler is distributed to its threads according to a low-level, process-specific software scheduler.

Implementation Results

We have used our developed tools to automatically generate wrappers based on the AADL pacemaker specification described in Section 2.2.3. The platform was instantiated and synthesized using Xilinx Embedded Development Kit 10.1.3 on a Virtex-5 VLX50T FPGA. We created both correct and faulty versions of low-criticality processes (in particular, the programmer process) to test the capability of the monitoring wrappers to reject faults. The design behaved according to specification, albeit more stringent testing would be required for certification.

Table 2.2 shows chip requirements in term of area (FPGA slices) for several modules. Since propagation delay is less of a concern in current FPGA chips, a single 125Mhz `system_clk` is shared among all wrappers. The VLX50T FPGA can support up to 11 additional clocks for processors and memories; in our pacemaker implementation, the CPU and all HW processors use separate `proc_clk` clocks also running at 125Mhz. The communication infrastructure can transfer up to 4 bytes per cycle, for a total bandwidth of 500MB/s; however, due to guard time, a maximum of 480 bytes can be transferred every 1us frame, resulting in an effective bandwidth of 480MB/s. Finally, we measured an OS footprint of 7648 bytes in instruction and 15234 bytes in data scratchpad.

2.2.5 Timing Isolation

As described in Section 2.2.2, each process certificate specifies a timing reservation for both computation and communication. More in details, each process expresses a *computation request* $\{e, p, D\}$, where e is the execution time, p the period and D the relative deadline specified in the AADL model. Furthermore, if a process defines N communication channels, then it expresses N *communication requests* of the form $\{e_i, p, D_i\}$, where p is the period of the process itself, D_i is the specified communication relative deadline for the i-th communication channel, and e_i represents its required transmission time in number of frames; e_i is obtained by dividing the data size by the communication infrastructure bandwidth and adding a constant term representing memory access delay and any per-transfer overhead (up to 40ns in our implementation), then rounding up to the frame size. All deadlines must be less or equal to periods.

Timing isolation is provided as follows: each computation request is treated as a periodic task to be scheduled on the assigned processor, and each communication request as a periodic task scheduled on the communication infrastructure. Our tool then generates all scheduling tables by computing *implicit-EDF* schedules [20]: for each CPU and the communication infrastructure, an Earliest-Deadline-First schedule is simulated for an entire hyperperiod (the minimum common multiple of process periods in number of frames), and the process executed during each frame is added to the corresponding table. This static scheduling approach has a key advantage for safety-critical systems: when verifying the correctness of the whole system, it is not necessary to model the scheduling algorithm. Instead, the generated scheduling tables can be used as input to the verification system. In this way as an example, a model checker would only need to check the system behavior for one hyperperiod, since the same exact schedule will be repeated thereafter. As a disadvantage, scheduling tables can potentially grow quite large if process periods are not homogenous; however, we argue that for most system designs, this is not the case (for example, in dataflow models,

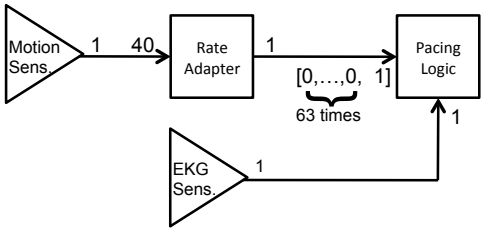


Figure 2.11: Example Dataflow Communication

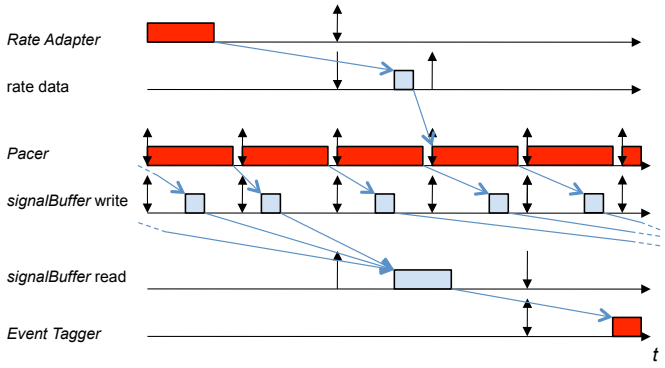


Figure 2.12: Example Schedule

process periods depend on the ratio between the number of sent and received tokens).

Note that our reservation model does not necessarily restricts the model of computation used by each application. In particular, both time-driven and event-driven activation models can be mapped to our specification, albeit in the second case it can lead to over-reservation. As an example, consider the pacemaker subsystem composed of the Motion Sensor, Rate Adapter, EKG Sensor and Pacing Logic. At a higher-level, this subsystem could be modeled as the cyclo-static dataflow represented in Figure 2.11, where the Pacing Logic consumes one token from the EKG Sensor every activation and one token from the Rate Adapter every 64 cycles. Based on this representation, periods of 7,810us for the Pacer process (including the EKG Sensor and Pacing Logic), 499,840us for the Rate Adapter and 12,496us for the Motion Sensor can be synthesized, which are within 0.1% of the frequencies (128Hz, 2Hz and 80Hz) specified in Section 2.2.3 for these processes.

A possible generated schedule for the Rate Adapter, Pacer and Event Tagger processes is shown in Figure 2.12, where up arrows represent periodic activation times and down arrows represent absolute deadlines. Slanted arrow are used to connect periodic process instances with the data they generate and consume. We represent a single instance of Rate Adapter and Event Tagger because their periods are significantly larger than the Pacer period (64 and 16 times larger respectively); furthermore, note that in reality communication reservations are much smaller than represented, since the size of the exchanged data is very small. Every period, the Event Tagger reads from the signalBuffer the data written by the previous 16 instances of Pacer. Note that the amount of written data is not constant: the Pacer writes EKG data every period, but a Shock Event is written only when the heart is paced. The Pacer communication request must be nevertheless sufficient to send both EKG and Shock data every period. To minimize end-to-end delay, the activation time of each communication task for outgoing data is set to be equal to the deadline of the corresponding process; vice versa, the deadline for a DMA read operation corresponds to the activation time of the next process instance. Furthermore, during mapping the designer can specify an initial activation time (*phase*) for each process; for example, in Figure 2.12 this is used to make sure that the activation time of Pacer corresponds with the deadline of the rate data sent by the Rate Adapter.

Goal	1.Compatibility	2.Effectiveness	3.Performance	4.Overhead	5.Simplicity	6.Reusability	7.Automation
PCI	very good	good	good	fair	poor	good	good
SoC	good	very good	very good	good	good	good	very good

Table 2.3: Assessment of Hardware Control Abstractions.

A final consideration is relative to queue size. Both the request queue and all FIFO queues in the interface module must be large enough to avoid overflow; if a queue is full, any incoming data is dropped. In general, required queue size depends on the internal behavior of a process (e.g. when it consumes data in the queue): therefore, the designer must specify minimum queue sizes in the AADL model and validate them for critical processes. As an example, consider the interaction between the Motion Sensor and the Rate Adapter. If the activation time for the Rate Adapter corresponds to the communication deadline for the Motion Sensor and furthermore the Rate Adapter consumes all data from the queue immediately upon activation, the minimum size of 40 samples is sufficient. However, if the time at which the Rate Adapter consumes the data is unknown, a size of up to 80 samples may be required (in general, it is possible to show that given periods p_i, p_j for a producer and consumer process respectively, a worst case size of $\lceil \frac{2p_j}{p_i} \rceil + 1$ samples is required assuming that the consumer reads all available data every period [100]).

2.3 Conclusions

We conclude by providing a qualitative assessment of how each of the two introduced control abstractions fares in term of the seven design goals delineated in the chapter introduction. Our considerations are summarized in Table 2.3. **(1)** A significant amount of effort has been spent in designing real-time bridges to be fully compatible with the operations of the underlying PCI standard. In particular, the real-time bridge is completely transparent from the point of view of the end application. We have also designed our platform interface module to be compatible with typical SoC communication interfaces. Adapting the Microblaze processor to run inside a hardware wrapper required very limited work, and we expect the same to hold for most available processor IPs. **(2)** Both abstractions are fully effective in controlling all communication activities by their components. Furthermore, monitoring hardware wrappers can also prevent unduly power consumption. **(3)** Both abstractions meet the performance requirements of their respective communication architectures by exploiting DMA transfers. However, experiments reveal that the Microblaze soft core employed in our real-time bridge does not offer sufficient performance to process complex data flows such as network packets at full speed. This issue could be easily solved by moving to a more performant CPU; for example, other Xilinx FPGA devices offer integrated PowerPC hard cores. **(4)-(5)** As detailed in Section 2.2.4, the monitoring hardware wrapper has limited overhead both in terms of area utilization and in terms of communication bandwidth. The RTL code of the interface module is rather simple, consisting of a few hundreds lines of VHDL. The implementation of the real-time bridge is significantly more complex, relying on a complete SoC hardware solution with the addition of

specialized host and FPGA drivers; verifying the correctness of the entire implementation would be a rather daunting proposition. Part of this complexity stems from the fact that we originally designed the real-time bridge only to provide a high-performance scheduling mechanisms for peripherals, with no consideration to functional isolation. However, the main issue is that the PCI protocol is inherently complex, and the real-time bridge must handle all its intricate details to be fully compatible. The current real-time bridge prototype requires two different chips to be implemented, a FPGA and a RAM bank, but we believe that they could be integrated in a motherboard with limited overhead in terms of space and power consumption. **(6)** Both abstractions provide a good amount of reusability. The core of the interface module can be reused across multiple communication infrastructures and processors, albeit the Processor Interface and Communication Interface module must be suitably adapted. The real-time bridge can be used across different peripherals, at the cost of adapting portions of the drivers, albeit porting it to a different COTS interconnection such as, for example, USB would require more significant modifications in the PLB bridges and drivers. **(7)** Finally, in both cases we have introduced additional scheduling components that communicate with all control abstractions to enforce a global scheduler, together with a schedulability analysis for simple interconnections. Additionally, our SoC toolchain can automatically generate functional and timing certificates based on a high-level description of the system requirements.

Chapter 3

Hardware Run-Time Monitoring

The goal of any safe mixed-criticality design is to prevent safety-critical applications from failing; low-criticality applications employing unreliable COTS components can be allowed to fail, albeit it is undesirable. As an example, consider the medical pacemaker described in Section 2.2.3. The pacemaker implements both a life-critical pacing component, as well as a variety of other applications such as data logging and remote communication and programming that are used for diagnosis. Unfortunately, formally verifying and/or certifying these non safety-critical applications is either extremely expensive or unfeasible: they typically make large use of both software and hardware Intellectual Properties (IPs) such as CPU cores and OSes that are very complex. As such, they must be allowed to fail. The problem is that active components, such as CPU and coprocessors, used by low-criticality applications must communicate with the life-critical application and share physical resources such as battery energy, memory and communication bandwidth; therefore, faults can propagate from a lower to a higher criticality application, making the whole system unsafe. In the same way, modern COTS peripherals in PC systems are particularly challenging. A DMA peripheral can directly communicate with all other elements in the system, including main memory and other peripherals, thus reducing the load on the CPU. On the other side, providing fault-containment becomes extremely hard: a misbehaving, low-criticality master peripheral could very well disrupt the entire system.

Based on the above discussion, our proposal to support functional isolation in critical embedded systems relies on *runtime monitoring*: the requirement specification for each unreliable active component is checked at runtime against its current observable behavior. If any violation is detected, then a suitable recovery action can be taken to restore the system to a safe state. The validity of the runtime monitoring approach has been proven in the field of software engineering by a large number of developed tools and techniques (see Section 3.6). However, applying runtime monitoring to our scenario poses new challenges, because active components can directly interact with the rest of the system without requiring any action by the CPU. Therefore, our monitoring solution must be able to detect and check all communication between active components in the system. Finally, runtime monitoring typically comes with an unforgivable price: runtime overhead. We can split such overhead in two components: 1) overhead due to

the observation and generation of relevant events 2) overhead due to running a monitor at each event to check if any property of the specification is violated. Both types of overhead tend to be unpredictable and thus unsuitable for real-time computation.

To overcome such problems, we propose a distributed monitoring technique which exploits our developed hardware control abstractions. The idea is to employ control abstractions to check all component communication and perform recovery actions, when necessary. Assuming "sniffing" data transfers does not add delay to the system, our solution prevents the first type of overhead. The second type of overhead is removed by running all monitors directly in the control abstractions, adding no runtime overhead to the CPU. Additionally, the system can run completely undisturbed as long as no recovery action is needed. The speed of modern COTS communication architectures rules out the possibility of a monitor software implementation; instead, all logic is implemented in hardware, using reconfigurable FPGAs in our prototypes. Finally, to show that a monitored system is safe, we need to prove that the monitoring logic monitors, indeed, the right properties. In our system, this is ensured by automatically synthesizing the monitoring logic from formal requirements specification, so that it is "correct by construction". In particular, we leverage on the Monitor Oriented Programming (MOP)[22] framework (see Section 3.1), which is highly extensible and supports multiple formalisms, by creating a new MOP instance: HardwareMOP.

Illustrative Example. An example of HardwareMOP can be seen in Figure 3.1, applied to the verification of a COTS peripheral. This example is a property used in the case study of Section 3.4 and related to the behavior of Counter 2, a counter on the PCI703A board we used in our experiments. This property, called SafeCounterModify, requires that any modification to cntr_cntrl2, the control register for Counter 2, happens only while the counter is not in use. This modification is captured by the cntrlMod event, because cntr_cntrl2 is at address X"220". The counter can be enabled/disabled by modifying bit 0 of cntr_cntrl2 (captured by the countEnable/countDisable events; "-" is the VHDL 'don't care").

The declarations section declares two monitor-local registers, cntrlCurrent and cntrlOld, and initializes them to 0. These registers will hold the current and previous values of the cntr_cntrl2 register. This allows us to repair the register when/if the property is *violated* by writing the old value to the register on the peripheral itself (the value_reg assignment), and forcing the current value the monitor stores to be the previous value, as can be seen in the violation handler section of the specification.

The extended regular expression pattern, in the ere section, matches any trace that consists of a cntr_cntrl2 modification, a disable of the counter, or an enable followed by a disable. The pattern is followed by *, allowing it to match repeatedly. The only way to violate this pattern, then, is to see a modification after an enable that is *not* followed by a disable first.

The implementation of the events, declarations, and the actions available to handlers is explained in Section 3.3.3. The formula/pattern implementation, and the use of handlers is explained in Section 3.3.4. The actual generated code is available in a technical report [73], or by running the online trial on our website [18].

```
declarations : {
  signal cntrlCurrent : STD_LOGIC_VECTOR(15 downto 0) := X"0000";
  signal cntrlOld : STD_LOGIC_VECTOR(15 downto 0) := X"0000";
}

event countDisable : memory write address = base1 + X"220"
                     dbyte value in "---------------0"
event cntrlMod : memory write address in base1 + X"220"
                 {
                   cntrlOld <= cntrlCurrent;
                   cntrlCurrent <= value(15 downto 0);
                 }
event countEnable : memory write address = base1 + X"220"
                    dbyte value in "---------------1"

ere: ((countEnable countDisable) + cntrlMod + countDisable)*

violation handler : {
  mem_reg <= '1';
  address_reg <= base1 + X"220";
  -- roll back to the previous cntr_cntrl2 value
  value_reg(15 downto 0) <= cntrlOld;
  cntrlCurrent <= cntrlOld;
  enable_reg <= "0011";
}
```

Figure 3.1: Example Property: SafeCounterModify

Key contributions. We provide three main contributions. First, in Section 3.2 we describe how hardware runtime monitors can be generated to monitor communication activities both for COTS peripherals on the PCI bus and in the SoC platform described in Section 2.2. Whenever an active component fails to conform to its functional specification, the associated hardware control abstraction can perform a *corrective* action to restore the system to a safe state. Second, in Section 3.3 we describe in details how formal properties are specified in our system. In the case of COTS peripherals, properties must be manually specified. In our SoC platforms, properties are derived as part of the formal certificate attached to a processor (see Section 2.2.2). Third, in Sections 3.4 and 3.5 we describe two relevant case studies to show the feasibility of our overall approach. We conclude by discussing related work in Section 3.6, and discussing future work in Section 3.7.

3.1 The MOP Framework

Monitoring-Oriented Programming (MOP) ([22] and citations there) is a formal framework for system development and analysis, in which the developer specifies desired properties using *definable* specification formalisms, along with code to execute when properties are violated or validated; it is important to note that a failure to confirm to the specification can be expressed as either the validation or violation of a property, see Section 3.4 for examples. Monitoring code is then automatically generated from the specified properties and integrated together with the user-provided code into the original system. MOP is a highly extensible and configurable runtime verification framework; currently there are two MOP instances: JavaMOP and HardwareMOP (the instance described in this chapter).

Property specifications consist of event definitions, which are instance dependent (e.g., pointcuts in JavaMOP

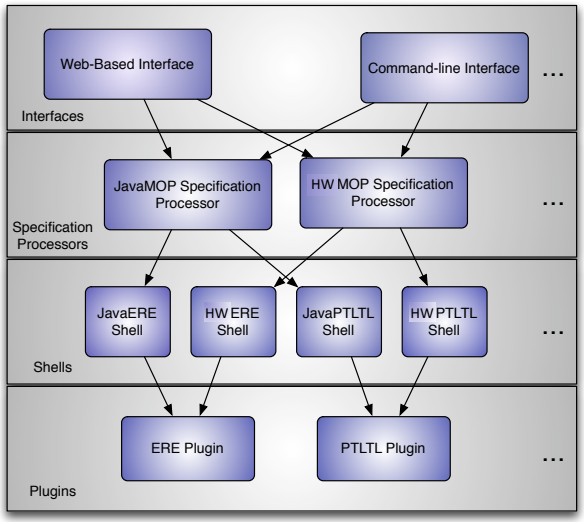


Figure 3.2: MOP Architecture.

and read/ write operations in HardwareMOP), and logical formulae or patterns, which are not. The user is allowed to extend the MOP framework with his/her own logics via *logic plugins* which encapsulate the monitor synthesis algorithms. This extensibility of MOP is supported by a layered architecture which separates monitor generation and monitor integration. By standardizing the protocols between layers, modules can be added and reused easily and independently. By providing language specific shells, logic plugins can be reused between several different MOP instances. A graphical representation of the architecture can be seen in Figure 3.2.

The formula or pattern designates which "traces" (observed series of events) are valid or invalid. Because Extended Regular Expression (ERE) and Past-Time Linear Temporal Logics (PTLTL) are the two plugins used in this work, we will describe which traces are valid or invalid for ERE patterns and PTLTL formulae. For EREs, valid traces are those which are strings in the language represented by the ERE, with events treated as the letters in the alphabet of the language. Neutral traces (which trigger no handlers) are prefixes of strings in the language, while violations are invalid strings. For PTLTL formulae, valid traces are any traces for which the formula evaluates to true, invalid traces are those for which the formula evaluates to false; there are no neutral traces. For more information on regular languages and temporal logic see [61] and [26], respectively.

3.2 Monitoring Logic

As described in Section 3.1, the MOP framework can automatically generate monitoring code in VHDL/Verilog Register Transfer Level (RTL), given a formal description of the system properties. A system block diagram for the generated code is shown in Figure 3.3. The system0, . . ., systemI, . . ., systemN blocks implement the monitoring

logic for each of N user specified properties. Each systemI block consists of two automatically generated modules: bus_interfaceI contains all logic that depends on the specific choice of communication interface, while monitorI contains all logic that depends on the formal language used to specify the property. This separation provides good modularity and facilitates code reuse. bus_interfaceI first receives decoded input signals and generates events, which are sequentialized by the events_sequentializer submodule (see Section 3.3.3), and then passed to monitorI using the events wires. monitorI checks whenever the formula for the I-th property is validated/violated and passes the information back to bus_interfaceI.

Different types of bus_interfaceI modules are employed to generate events for read/write transactions in the case of CPU or HW processors in the SoC platform of Section 2.2 and in the case of PCI peripherals. In the former case, input decoded signals to bus_interfaceI are provided by the interface module described in Section 2.2.4. In the latter case, we designed a prototype monitoring device based on a Xilinx ML455 board [109] to monitor peripheral transactions. The board is outfitted with a Virtex-4 FPGA and is can be plugged into a standard 3.3Volts PCI/PCI-X socket; the device continuously "sniffs" all ongoing activities on the bus, and it is therefore able to monitor communication (read/write transactions) for all other peripherals located on the same PCI bus segment. In alternative, monitoring logic could be added to the real-time bridges described in Section 2.1 to decode peripheral reads/writes in the bridge local memory and forward this information to the monitoring logic.

Whenever the monitoring logic detects a failure to meet the formal specification, a recovery action is executed using strategies based on the detected error and the capability of the hardware control abstractions; recovery actions are specified by the user in a recovery handler that is executed whenever a given property is either validated or violated. Our system supports the following recovery strategies. **(1)** The current data transactions is rejected. In our SoC platform, this is implemented through a reset signal (see Figure 3.3) which is propagated back to the interface module. Note that only outgoing data transfers can be rejected; incoming data is always accepted. By checking all outgoing transfers, we always make sure that any data propagated on the communication infrastructure conforms to specified certificates. Hence, incoming data must also conform to system specification. Furthermore, since timing isolation is always enforced, DMA read operations can not steal resources or change the state of other processes. Unfortunately, our prototype monitoring device for PCI bus does not allow us to reject transactions: when a transaction is seen on the bus, which does not conform to the specification, that specific bus transaction can not be prevented from propagating to the rest of the system. This issue could be solved by integrating the monitoring logic in a real-time bridge: in this way, the real-time bridge could simply discard any data that causes a failure in the specification. **(2)** A write transaction can be sent to another element in the system (main memory for peripherals on the PCI bus, or a message queue or external memory in our SoC platform). Our implemented PCI monitoring device, the real-time bridge and the SoC interface module all implements DMA Engines that can be used to perform the write operation. Oftentimes, this strategy is enough to force a peripheral back into a consistent state, even when recovery strategy (1) is not allowed by the available control abstraction. For example, consider a common type of error, where the driver fails to validate

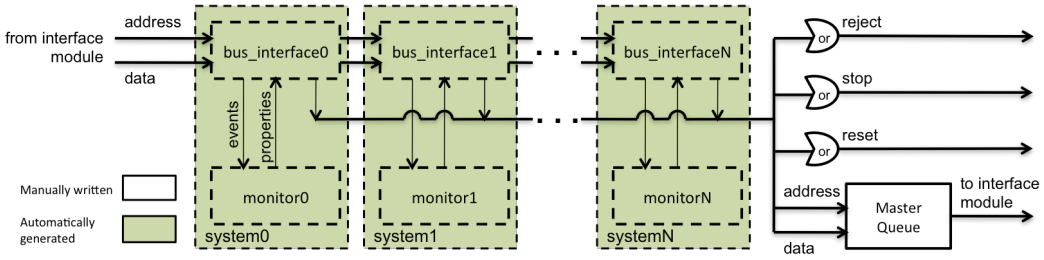


Figure 3.3: Monitoring Logic Block Diagram

some input from the user and as a result writes an invalid value to a register in the peripheral. We can recover by rewriting the register with a valid value. Furthermore, note that in our SoC platform, monitor requests take precedence over all other data transfer requests, but they can still be carried out only during a frame assigned to the process by the Communication Scheduler. **(3)** The faulty component (HW processor, CPU process or PCI peripheral) can be stopped. Our PCI monitoring device includes a peripheral gate that can disconnect any monitored peripheral from the bus. In the case of a real-time bridge, communication can be stopped by halting the operation of the DMA Engine. On our SoC platform, this is achieved by stopping the `proc_clk` in a HW processor; for a software process, the output of the CPU Scheduler is instead altered so that the process is never executed. **(4)** Finally, in our SoC platform a faulty process can be reset. The reset functionality depends on the design of the processor that executes the faulty process. HW processors implemented on FPGA use synchronous logic and are provided with a reset signal from the control module. In our Microblaze implementation, the control module can interrupt the CPU and signal the reset action. Xilkernel can then kill the process and restart it.

It is important to notice that in the current implementation the time elapsed from any event that triggers a validation/violation to executing the corresponding recovery handler is at most 4 clock cycles. For our PCI monitoring device, this time is short enough to execute a recovery action before a faulty peripheral is allowed to start a new transaction, as PCI arbitration overhead prevents a peripheral from transmitting immediately. For our SoC interface module, the presence of the transfer queue ensures that a faulty transaction can be rejected before being transmitted on the interconnection.

3.3 Property Specification

Properties are specified using a domain specific event syntax, and formulae or patterns written in the logic of a particular plugin. Additional monitor states can also be declared using the declarations section. The violation handler and validation handler sections allow for arbitrary code to be executed on the occurrence of a violation or validation, respectively. An example of how they are used can be seen in Figure 3.1. Currently, we have support for the extended regular expression (ERE) and past-time temporal logic (PTLTL) MOP Plugins, and adding most of the others will

require a minimal amount of work, as only the monitor component changes from one logical specification formalism to another. This means that properties may be specified, formally, using an ERE pattern or a PTLTL formula.

A property is implemented in two main modules, a bus_interface, which generates logical events from bus traffic and handles monitor recovery, and a monitor implementing a property specification in hardware. A more detailed description will be given below; since events are different in the case of PCI peripherals and of SoC processors, we describe event specification in two different subsections.

3.3.1 PCI Transaction Events

A formal description of the event syntax (using Backus Naur Form (BNF) [51] extended with $[p]$ and $\{p\}$, denoting zero or one repetitions of p and zero or more repetitions of p, respectively) can be seen below:

$$
\begin{array}{rcl}
\langle \textit{Event} \rangle & ::= & \text{“}event\text{”}\,\langle \textit{ID} \rangle : \langle \textit{Expression} \rangle \\
\langle \textit{Expression} \rangle & ::= & \langle \textit{MemoryOrI/O} \rangle\,\langle \textit{ReadOrWrite} \rangle\,\text{“}address\text{”}\,\text{“}=\text{”} \\
 & & \langle \textit{ArithmeticExp} \rangle\,\text{“}value\text{”}\,[\text{“}not\text{”}]\,\text{“}in\text{”}\,\langle \textit{Range} \rangle \\
 & & [\langle \textit{Action} \rangle] \\
 & | & \langle \textit{MemoryOrI/O} \rangle\,\langle \textit{ReadOrWrite} \rangle\,\text{“}address\text{”}\,\text{“}in\text{”} \\
 & & \langle \textit{Range} \rangle\,[\text{“}\{\text{”}\,\langle \textit{Action} \rangle\,\text{“}\}\text{”}] \\
 & | & \text{“}interrupt\text{”}\,[\langle \textit{Action} \rangle] \\
\langle \textit{MemoryOrI/O} \rangle & ::= & \text{“}memory\text{”} \quad | \quad \text{“}io\text{”} \\
\langle \textit{ReadOrWrite} \rangle & ::= & \text{“}read\text{”} \quad | \quad \text{“}write\text{”} \\
\langle \textit{Action} \rangle & ::= & \text{“”}\,\langle \textit{Arbitrary VHDL code} \rangle\,\text{“”} \\
\langle \textit{Range} \rangle & ::= & \langle \textit{ArithmeticExp} \rangle\,[\text{“,”}\,\langle \textit{ArithmeticExp} \rangle] \\
\langle \textit{ArithmeticExp} \rangle & ::= & \langle \textit{Number} \rangle \quad | \quad \langle \textit{ID} \rangle \\
 & | & \langle \textit{ArithmeticExp} \rangle\,\text{“}+\text{”}\,\langle \textit{ArithmeticExp} \rangle \\
 & | & \langle \textit{ArithmeticExp} \rangle\,\text{“}-\text{”}\,\langle \textit{ArithmeticExp} \rangle \\
 & | & \langle \textit{ArithmeticExp} \rangle\,\text{“}\&\text{”}\,\langle \textit{ArithmeticExp} \rangle \\
\langle \textit{Number} \rangle & ::= & \langle \textit{VHDL number or bitstring} \rangle \\
\langle \textit{ID} \rangle & ::= & \langle \textit{Capital or lower case letter} \rangle\,\{\langle \textit{LetterOrDigit} \rangle\} \\
\langle \textit{LetterOrDigit} \rangle & ::= & \langle \textit{Capital or lower case letter} \rangle \quad | \quad \langle \textit{Digit} \rangle
\end{array}
$$

There are three basic types of events for PCI peripherals: I/O accesses, memory accesses, and interrupts. The PCI standard defines both a I/O address space, used mostly by legacy peripherals, and a memory space, where main memory and most modern peripherals are mapped. It is important to distinguish between I/O and memory events because they require different enable functionality and different read/write signals. We show uses of interrupt events in [73]. I/O and memory events must specify at least an address, which may be an arithmetic expression over identifiers,

VHDL numbers, addition, subtraction, and concatenation, and whether the event is a read or a write. An I/O or memory event may also specify a value range, which is the value of the address read or written by the bus transaction. Ranges can consist of a single arithmetic expression, or a pair of comma separated arithmetic expression denoting the minimum and maximum values that may trigger the event (thus, ranges are inclusive). Value ranges must also specify a size, byte, dbyte (16 bits), or qbyte (32 bits), so that the correct comparison code and byte enables can be generated (values smaller than a byte require masking the proper bits). Address ranges are used in events that *do not* have specified value ranges. The reason for this is that when a value range is specified, the code generator must generate the proper byte enables based on address alignment, and alignment does not make sense for ranges. Address ranges are useful for some properties, e.g. a property that monitors accesses to a certain buffer in memory.

3.3.2 SoC Processor Events

A formal description of the event syntax can be seen below, where ⟨*ID*⟩ represents a symbolic name, ⟨*Number*⟩ a constant number, and ⟨*ArithmeticExp*⟩ an arithmetic expression:

$$
\begin{array}{rcl}
\langle \textit{Event} \rangle & ::= & \langle \textit{ID} \rangle : \langle \textit{Expression} \rangle \\
\langle \textit{Expression} \rangle & ::= & \langle \textit{ReadOrWrite} \rangle \text{“}in\text{”} \langle \textit{ID} \rangle [\langle \textit{Action} \rangle] \\
& | & \langle \textit{ReadOrWrite} \rangle \text{“}at\text{”} \langle \textit{ID} \rangle [\text{“} + \text{”} \langle \textit{Number} \rangle] \\
& & [\text{“}value\text{”} [\text{“}not\text{”}] \text{“}in\text{”} \langle \textit{Range} \rangle] [\langle \textit{Action} \rangle] \\
\langle \textit{ReadOrWrite} \rangle & ::= & \text{“}read\text{”} \quad | \quad \text{“}write\text{”} \\
\langle \textit{Action} \rangle & ::= & \text{“}\langle \textit{Arbitrary VHDL code} \rangle\text{”} \\
\langle \textit{Range} \rangle & ::= & \langle \textit{ArithmeticExp} \rangle [\text{“,”} \langle \textit{ArithmeticExp} \rangle]
\end{array}
$$

All events are read/writes from/to either a message queue or a data element in external memory, identified by its name in the AADL model. Normally, an event is triggered whenever any word of a data element is changed. However, the designer can specify a numerical address inside the data element, in which case the event is triggered only when that specific memory location is read/written. This can useful to access individual components of a complex data structure. In this case, the designer can also specify a desired value range. Ranges can consist of a single arithmetic expression, or a pair of comma separated arithmetic expressions denoting the minimum and maximum values that can trigger the event. Since all transfers in the system are multiple of the word size, values are always assumed to be 32-bits.

3.3.3 The bus_interface Module

The code for the declarations, and handler sections is copied verbatim into the VHDL module defining the bus_interface. Because of this copying, the code must be written in VHDL. The events are expanded to combinatoric statements implementing the specified logic. The output of the combinatoric statements is assigned to an events wire vector, which

	Write Interface
address_reg	address
value_reg	value
enable_reg	PCI byte enables
io_reg	write request in I/O space
mem_reg	write request in memory space
serial_reg	ASCII value to serial output
stop_reg	stop peripheral

Table 3.1: Recovery handler registers, PCI

	Data Transfer Interface
address_reg	address
value_reg	value
execute_reg	start transfer
reject_reg	reject transfer
stop_reg	stop process
reset_reg	reset process

Table 3.2: Recovery handler registers, SoC

is connected to the monitor module through an event_sequentializer submodule. Each index in the bus corresponds to the truth value of a specific event, numbered with the 0'th index as the first event, and the n'th index as the n'th event from top to bottom in the specification. This ordering is important, because it directs the event linearization performed by the event_sequentializer submodule.

The event_sequentializer is necessary because the logical formalisms expect linear, disjoint events. The event_sequentializer takes coincident events and sends them to the monitor in subsequent clock cycles, in ascending index order, using the seq_events wire vector. Therefore, if events(0) and events(3) occur in the same cycle, the monitor will see 0 followed by 3. To see why simultaneous events are possible, consider, again, the example in Figure 3.1. The cntrlMod event is asserted whenever the cntr_cntrl2 register (base1 + X"220") is written. Because both the countEnable and countDisable events require writes to the same address as the cntrlMod event, any time countEnable or countDisable are triggered, a cntrlMod is also triggered. As the property tries to enforce the policy that all modifications happen when the counter is not enabled, we must serialize events such that cntrlMod happens *after* a countDisable and *before* a countEnable. The ordering of events in Figure 3.1, is consistent with this, because countDisable is listed before cntrlMod, which is listed before countDisable.

The violation handler is placed in the module such that it is only executed if the monitor module denotes that the property has been violated. The situation is similar for a validation handler, save that it is executed only when the formula or pattern is validated. Currently, recovery actions in the handlers are specified as a list of concurrent VHDL statements. While it could be possible to define a new formalism for recovery specification, we believe it would not be very beneficial for two main reasons: the formalism would have to be specific to HardwareMOP; specifying correct recovery actions inevitably requires a deep understanding of the monitored hardware, therefore VHDL seems well suited to the task. As can be seen in the Figure 3.3, the monitor module reports the validation, violation, or neutral state of the monitored property, via the properties wire vector, to the bus_interface module. Several actions are available in validation and violation handlers. Aside from manipulating any local state of the monitor (such as the write to cntrlCurrent in Figure 3.1), the bus_interface module makes available several registers which can be used

used to execute the recovery actions detailed in Section 3.2. The registers are summarized in Tables 3.1, 3.2 for PCI peripherals based on our monitoring device and for processors in our SoC platform, respectively. In the latter case, note that symbolic names can be used in place of addresses in the AADL specification.

As can be seen in Figure 3.1, we perform a memory write to the cntr_cntrl2 register of its previous value. The address_reg is used to denote the address of cntr_cntrl2 (base1 + X"202"), while the value_reg is set to the old value of cntr_cntrl2, the mem_reg is asserted to tell the PCI bus that the write performed is a memory write, and the byte enables are set to "0011" to denote that the lower two bytes must be written.

3.3.4 The monitor Module

The monitor module is responsible for monitoring the property given serialized events. It encompasses the logic of the formula, and it is the only portion of our system dependent on the logical formalism used.

Extended Regular Expressions. Extended regular expressions (EREs) are the normal regular expressions [61], extended with negation. The same plugin used for JavaMOP's [22] EREs is used to transform the provided ERE to a minimized deterministic finite automata (DFA) defined in generic code. We convert the generic code to Verilog. The current state of the DFA is kept in a register. On each clock cycle, the current state of the DFA and the event are consulted to see if the property is violated or validated, and what state to transition to. Violations of EREs are tricky, because, if used normally, a DFA, once it reaches a violation state, will report a violation every event (because there is no valid transition out of the violation state). We chose to reset the DFA to the initial state whenever a violation is encountered, to avoid this problem. ERE pattern is as follows:

$$
\begin{aligned}
\langle \textit{Pattern} \rangle \quad ::= \quad & \text{“}\textit{epsilon}\text{”} \quad | \quad \langle \textit{Event Name} \rangle \\
| \quad & \text{“}\sim\text{”}\langle \textit{Pattern} \rangle \quad | \quad \langle \textit{Pattern} \rangle \text{“}*\text{”} \\
| \quad & \langle \textit{Pattern} \rangle \text{“}+\text{”}\langle \textit{Pattern} \rangle \quad | \quad \langle \textit{Pattern} \rangle \langle \textit{Pattern} \rangle
\end{aligned}
$$

“epsilon” is the empty string, “ $\sim$ ” is negation, “ $*$ ” is zero or more repetitions, “+” is logical *or*, and $\langle \textit{Pattern} \rangle$ $\langle \textit{Pattern} \rangle$ represents concatenation. As an example, the generated code for the property described in the chapter introduction is shown below.

```
module monitor0(clk, rst, empty, events, properties);
   parameter NUM_EVENTS = 3;
   parameter NUM_PROPERTIES = 2;
   input clk;
   input rst;
   input empty;
   input [NUM_EVENTS-1:0] events;
```

```
   output [NUM_PROPERTIES-1:0] properties;
   reg [NUM_PROPERTIES-1:0]  properties_reg;
   reg [1:0] state;
   assign properties = properties_reg;
   always @(posedge clk or posedge rst) begin
      if (rst) begin
         properties_reg <= 0;
         state <= 0;
      end else begin
         if (events[NUM_EVENTS-1:0] != 0) begin
         // properties 0 == nothing, properties 1 == success
         // properties 2 == failure
            case (state)
               0: begin
                     state <= (events[2])?1:(events[1])?0:(events[0])?0:0;
                     properties_reg <= (events[2])?0:(events[1])?1:
                        (events[0])?1:2;
                  end
               1: begin
                     state <= (events[0])?0:0;
                     properties_reg <= (events[0])?1:2;
                  end
               default : begin state <= 0; properties_reg <= 2; end
            endcase
         end
         else //if events[NUM_EVENTS-1:0] == 0
            properties_reg <= 0;
      end
   end
endmodule
```

Past-time Linear Temporal Logic. Past-time Temporal Linear Logic (PTLTL) [26] extends normal propositional logic with *temporal* operators. We modified the PTLTL plugin used in JavaMOP to make it more suitable for implementation as a logic circuit. The original, generic code output by the plugin used a number of sequential assignments to an array of truth values. We take this sequential code and, using back substitution, change the sequential code into

a series of parallel assignments. The resulting assignments are *entirely* parallel, allowing the operation of the monitor to be contained within a single clock cycle. A more in depth explanation of this transformation is omitted, but will appear in an upcoming technical report on PTLTL. The syntax for PTLTL formulas is as follows:

$$
\begin{aligned}
\langle \textit{Formula} \rangle \quad ::= \quad & \textit{“true”} \mid \textit{“false”} \mid \langle \textit{Event Name} \rangle \\
\mid \quad & \textit{“not”} \langle \textit{Formula} \rangle \mid \langle \textit{Formula} \rangle \textit{“and”} \langle \textit{Formula} \rangle \\
\mid \quad & \langle \textit{Formula} \rangle \textit{“or”} \langle \textit{Formula} \rangle \\
\mid \quad & \langle \textit{Formula} \rangle \textit{“implies”} \langle \textit{Formula} \rangle \\
\mid \quad & \text{“}[*]\text{”} \langle \textit{Formula} \rangle \mid \text{“}\langle * \rangle\text{”} \langle \textit{Formula} \rangle \\
\mid \quad & \text{“}(*)\text{”} \langle \textit{Formula} \rangle \mid \langle \textit{Formula} \rangle \text{“}S\text{”} \langle \textit{Formula} \rangle
\end{aligned}
$$

“not”, *“and”*, *“or”*, and *“implies”* are the ordinary logic operators. “$[*]$”, “$\langle * \rangle$”, “$(*)$”, and *“S”* are temporal operators denoting always in the past, eventually in the past, previously, and since, respectively.

As an example of the transformation to efficiently monitorable code, consider the PTLTL formula grant implies $\langle$*$\rangle$ request. This formula states that if a grant of some resource occurs, then at some point in the past there must have been a request[1]. This results in sequential code: b[0] := request or b[0]; output(not grant or b[0]);. b is the array of truth values used by the monitoring algorithm; each truth value becomes a single bit flip flop in the FGPA implementation. The statement output tells us what the output state of the monitor is, i.e. at a given time event arrival, the original formula is true if b[0] is true. Because this is sequential code, it is significant that output is the last statement (it need not necessarily be last). grant implies $\langle$*$\rangle$ request is changed to not grant or $\langle$*$\rangle$ request by boolean simplification. If we evaluate the sequential code for a simple trace request grant, we see that when request arrives b[0] is assigned the value true, regardless of the previous value of b[0], and that true is output. If the output statement were first, the output would be false on the first request event. When the grant arrives the monitor again outputs true because b[0] is true.

In order to transform this into a series of parallel assignments we need substitute the rhs R of an assignment statement $b[i] := R$ into all assignments $b[j] := R'$ *after* $b[i] := R$ such that $b[i] \in R'$. The reason for this is that with parallel assignments all rhs are evaluated before any assignments occur. The final parallel assignment code (denoted by = rather than := is b[0] = request or b[0]; output(not grant or request or b[0]) . As we can see, request or b[0] has been substituted for the original reference to b[0]; the remaining b[0] contains the value from the previous event.

A design decision relating to both logics we have implemented, and all future logics, is that properties cannot be violated or validated before an event arrives. Without this assumption, the example ERE property would be valid at start up. This creates a problem: to correctly trigger recovery actions in the bus_interface module, we require that the properties wire be set to 1/2 (for a validation/violation respectively) for only one clock cycle. The solution we adopted is simply to set properties to zero when no event is detected. An additional problem is that without the assumption, a single event in ERE could cause a violation followed immediately by a validation (since we reset the monitor on

[1]This property is over simplified: multiple grant's are allowed for one request

violation) in the same clock cycle. This could in turn trigger both a validation and violation handler at the same time, which is something we can not support. JavaMOP has the same functionality, but in JavaMOP it is due to the fact that a monitor does not *exist* before the first event, whereas in HardwareMOP, the monitor exists as soon as the FPGA is configured.

3.4 Case Study: The PCI703A ADC/DAC Board

In this section, we show how our runtime monitoring technique can be applied to a concrete case by providing specification and runtime experiments for a specific COTS peripheral, the PCI703A board [27]. PCI703A is a high performance Analog-to-Digital/Digital-to-Analog Conversion (ADC/DAC) peripheral for the PCI bus. In particular, it can perform high-speed, 14-bits precision ADC at a rate of up to 450,000 conversions/s, and transfer data to main memory in bus master mode. At the same time, the peripheral is simple enough that we were able to carefully check all provided hardware manuals and to manually inspect its Linux driver; specifying formal properties for a peripheral clearly requires a deep understanding of its inner working. In our proposed model, the peripheral's manufacturer is responsible for writing the runtime specification. In this sense, the formal specification can be thought of as a correctness certification provided by the manufacturer, as long as the user employs a monitoring device and recovery actions can be proved to restore the system to a safe state.

To better mimic what we think would be a typical process for a COTS manufacturer, we produced a requirement specification for the PCI703A in two steps. First, we prepared a detailed description of the communication behavior of the peripheral in plain English. Then, we converted this informal description into a formal set of events and formulae for HardwareMOP. Inspection of the driver revealed two software faults, both of which can cause errors that are detected and recovered by the monitoring device. While in this case we could have prevented errors by simply removing the faults, we argue that drivers for more complex peripherals can be thousands of lines long and neither code inspection nor testing is sufficient to remove all bugs. We further injected additional faults in the driver to test all written formal properties. It would have been nice to also show recovery for hardware faults, but we did not find any in the tested peripheral and injecting faults in the hardware is difficult. A list of both informal and formal properties can be found at [73]. In what follows, we first provide an overview of PCI703A and then we detail properties for an example subsystem, a counter used in the ADC process. The example is particularly instructive as we show how a small but representative set of properties is able to catch one of the aforementioned driver bug.

A block diagram for the PCI703A is shown in Figure 3.4. The bus slave logic implements two memory address blocks in BAR0 and BAR1, used for conversion data and control registers, respectively; the corresponding base addresses are written in base0 and base1 in the monitoring device. The ADC Control and DAC Control blocks control the ADC/DAC operations and write/read data into internal FIFOs. The DMA Control block can be programmed to move data between each FIFO and main memory using bus master functionality. Finally, the Counter Timers block implements four counters. Counter 0 and 1 are user programmable and can be used either for debugging purposes or

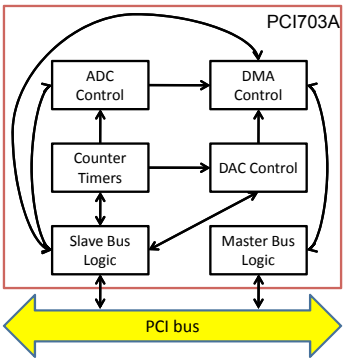


Figure 3.4: PCI703A Diagram.

to trigger a DA conversion. Counter 3 is also user programmable and produces an external output. Finally, Counter 2 is not meant to be user programmable; it is to be used exclusively to generate the clock for AD conversions. The C user library provided with the driver exports an ADConfig function used to configure ADC Control and the associated Counter 2. The library also provides a CTConfig function to be used to configure the user counters; unfortunately, under Linux the function can also be used to change the configuration of Counter 2. This is a problem, as any user in the system could erroneously or maliciously change Counter 2 while an ADC is in progress.

Three 16-bits control registers are relevant to our discussion: cntr_cntrl2 (at hexadecimal location 220 relative to BAR1), cntr_divr2 (228), and adc_cntrl (300). Bit 0 of cntr_cntrl2 determines whether Counter 2 is enabled, and bits 2-1 determine its clock source (either 20Mhz or 100Khz); when the counter is enabled, it first loads the content of cntr_divr2 and then starts counting down at the selected frequency. When it reaches zero, the value of cntr_divr2 is reloaded, a clock signal is sent to ADC Control, and finally if bit 4 of cntr_cntrl2 is set, an interrupt is generated. Register adc_cntrl controls the behavior of ADC Control; in particular, bit 0 enables/disables the ADC process and bits 2-1 determine the clock source, with a value of ”00” indicating that Counter 2 is used.

We express three requirements:

Requirement 1 *Bit 4 of cntr_cntrl2 should never be set. While the functionality is relevant for Counters 0,1, in the case of Counter 2 setting bit 4 would cause the generation of spurious interrupts that increase load on the driver.*

Requirement 2 *If the ADC is using Counter 2, and the clock source for Counter 2 is set to 20 Mhz, then the value of cntr_divr2 must be at least 45 to avoid violating the maximum conversion speed of the peripheral.*

Requirement 3 *If the ADC is active and using Counter 2, then Counter 2 must also be active; furthermore, while Counter 2 is active no change to the counter configuration is allowed.*

Requirements 1-3 are able to catch the driver bug in the sense that an invalid counter configuration can not be set before starting the ADC, and furthermore while the ADC is active no counter modification is allowed. We wrote four

```
declarations : {
  signal cntrlCurrent : STD_LOGIC_VECTOR(15 downto 0) := X"0000";
  signal cntrlOld : STD_LOGIC_VECTOR(15 downto 0) := X"0000";
}

event cntrlMod : memory write address in base1 + X"220"
                 {
                   cntrlOld <= cntrlCurrent;
                   cntrlCurrent <= value(15 downto 0);
                 }
event setBit4 : memory write
                address = base1 + X"220"
                dbyte value in "-----------1----"

ere: setBit4

validation handler : {
  mem_reg <= '1';
  address_reg <= base1 + X"220";
  -- roll back to the previous cntr_cntrl2 value
  value_reg(15 downto 0) <= cntrlOld;
  cntrlCurrent <= cntrlOld;
  enable_reg <= "0011";
}
```

Figure 3.5: InterruptFix Specification

```
declarations : {
  signal clkSrc : STD_LOGIC_VECTOR(15 downto 0) := X"0000";
  signal src : STD_LOGIC_VECTOR(15 downto 0) := X"0000";
}

event divrBad: memory write address = base1 + X"228"
               dbyte value in 0,44
event divrGood: memory write address = base1 + X"228"
                dbyte value in 45,65535
event clkSrcSet : memory write address in base1 + X"300"
                  { clkSrc <= value(15 downto 0); }
event srcSet :  memory write address in base1 + X"220"
                { src <= value(15 downto 0); }
event countEnable : memory write address = base1 + X"220"
                    dbyte value in "---------------1"

ere: (divrBad (clkSrcSet + srcSet)* countEnable)*

validation handler : {
  if (clkSrc(2 downto 1) = "01") and
    (src(2 downto 1) = "00") then
    mem_reg <= '1';
    address_reg <= base1 + X"228";
    --set cntr_divr2 to 45
    value_reg(15 downto 0) <= X"2D";
    enable_reg <= "0011";
  end if;
}
```

Figure 3.6: SafeConversionSpeed Specification

(five including the example of Figure 3.1) formal properties to capture the requirements:

InterruptFix. The InterruptFix specification is the formalization of Requirement 1, and can be seen in Figure 3.5. Because we do not want the 4th bit set, we simply monitor the pattern setBit4, an event which corresponds to setting the 4th bit. We perform recovery when the pattern is validated by overwriting cntr_cntrl2 with the last valid value, similarly to SafeCounterModify in Figure 3.1.

SafeConversionSpeed. The SafeConversionSpeed specification is the formalization of Requirement 2, and can be seen in Figure 3.6. For this property we chose to show how event side effects can be used in handlers as part of checking that a property has been validated/violated. When the clkSrcSet or srcSet events are triggered, meaning that the cntr_cntrl2 or adc_cntrl registers have been modified, respectively, we store the value written to the register in monitor local registers (e.g., src <= value(15 downto 0)). The pattern specifies that the cntr_divr2 be set to a bad value (less than 45), followed by any number of updates to cntr_cntrl2 or adc_cntrl, followed by the enabling of the counter. If cntr_divr2 is set to a value larger than 44, the pattern will be violated, and the monitor will be reset. This means that the validation handler will be executed only when then value of cntr_divr2 is too low for safe conversion, but regardless of whether or not the board is actually using Counter 2. The handler then checks that it is, in fact using Counter 2, and that Counter 2 is using the 20Mhz source, before performing the recovery: setting cntr_divr2 to a valid value (45).

NoDisableWhileConverting. The NoDisableWhileConverting specification is the formalization of part of Requirement 3, and can be seen in Figure 3.7. This could have been written in a similar manner to SafeConversionSpeed, i.e., using event side effects to store current register values and checking them in the handler. We decided to use a fully formal specification, that defines events for setting the registers to good or bad values. The formula itself specifies

```
declarations : {
  signal cntrlCurrent : STD_LOGIC_VECTOR(15 downto 0) := X"0000";
  signal cntrlOld : STD_LOGIC_VECTOR(15 downto 0) := X"0000";
}

event countEnable : memory write address = base1 + X"220"
                    dbyte value in "---------------1"
                    {
                      cntrlOld <= cntrlCurrent;
                      cntrlCurrent <= value(15 downto 0);
                    }
event countDisable : memory write address = base1 + X"220"
                     dbyte value in "---------------0"
                     {
                       cntrlOld <= cntrlCurrent;
                       cntrlCurrent <= value(15 downto 0);
                     }
event clkSrc2Good : memory write address = base1 + X"300"
                    dbyte value in "-------------01-"
event clkSrc2Bad : memory write address = base1 + X"300"
                   dbyte value not in "-------------01-"
event adcEnable : memory write address = base1 + X"300"
                  dbyte value in "---------------1"
event adcDisable : memory write address = base1 + X"300"
                   dbyte value in "---------------0"

ptltl: ( ((not adcDisable) S adcEnable) and
       ((not clkSrc2Bad) S clkSrc2Good) )
       implies
       ((not countDisable) S countEnable)

violation handler : {
  mem_reg <= '1';
  address_reg <= base1 + X"220";
  -- roll back to the previous cntr_cntrl2 value
  value_reg(15 downto 0) <= cntrlOld;
  cntrlCurrent <= cntrlOld;
  enable_reg <= "0011";
}
```

Figure 3.7: NoDisableWhileConverting Specification

```
declarations : {
  signal divrCurrent : STD_LOGIC_VECTOR(15 downto 0) := X"0000";
  signal divrOld : STD_LOGIC_VECTOR(15 downto 0) := X"0000";
}

event countDisable : memory write address = base1 + X"220"
                     dbyte value in "---------------0"
event divrMod : memory write address in base1 + X"228"
                {
                  divrOld <= divrCurrent;
                  divrCurrent <= value(15 downto 0);
                }
event countEnable : memory write address = base1 + X"220"
                    dbyte value in "---------------1"

ptltl: (divrMod) and (*)((not countDisable) S countEnable)

validation handler : {
  mem_reg <= '1';
  address_reg <= base1 + X"228";
  -- roll back to the previous cntr_divr2 value
  value_reg(15 downto 0) <= divrOld;
  divrCurrent <= divrOld;
  enable_reg <= "0011";
}
```

Figure 3.8: SafeDivrModify Specification

that, if the ADC is enabled, and clkSrc2 is good, meaning that Counter 2 is being used to time the ADC, then Counter 2 must be enabled. The part of the formula before the implies keyword, states that the ADC is enabled and the ADC clock source is Counter 2, the second half of the formula is the requirement that Counter 2 not be disabled. The formula is true when correct behavior is exhibited, so we use a violation handler for the recovery action, which again is simply to set cntr_cntrl2 to the last valid value.

SafeDivrModify. The SafeDivrModify specification is the formalization of part of Requirement 3, and can be seen in Figure 3.8. In conjunction with NoDisableWhileConverting and SafeCounterModify (from the chapter introduction), all of requirement 3 is covered. This specification ensures that cntr_divr2 is not modified while Counter 2 is enabled. This property is the same as SafeCounterModify from Figure 3.1, save that we are ensuring that cntr_divr2 is not modified, rather than cntr_cntrl2. We also used PTLTL rather than ERE, to show how two very similar properties look in both logics. These could be collapsed into one specification, but it would make recovery more complicated, because we only want to roll back the register that was actually modified (cntr_cntrl2 or cntr_divr2). The formula itself states that if cntr_divr2 has been modified and the counter has not been disabled since the last time it was enabled, than we must recover. Unlike SafeCounterModify we use a validation rather than a violation handler, because the formula

```
checkPacer   : write at Pacer.parameters       value in X"40000000"
checkRate    : write at RateAdapter.parameters value in X"40000000"
successPacer : read  at Programmer.response    value in X"80000000"
successRate  : read  at Programmer.response    value in X"20000000"
failurePacer : read  at Programmer.response    value in X"40000000"
failureRate  : read  at Programmer.response    value in X"10000000"
commitPacer  : write at Pacer.parameters       value in X"80000000"
commitRate   : write at RateAdapter.parameters value in X"80000000"
```

Figure 3.9: Programmer EventList

```
Formula:
1. ptltl : successPacer and <*>successRate and
2.   (*)( not (not(successRate) S successPacer) ) and
3.   (*)( not (not(successRate) S failureRate) )

Validation handler:
{
  address_reg <= Pacer.parameters;
  value_reg <= X"8000000";
  execute_reg <= '1';
}
```

Figure 3.10: SendPacerCommit Property

```
Formula:
1. ptltl : commitRate or commitPacer or (
2.   (checkRate or checkPacer) and
3.   (*)(
4.     (not(successRate or failureRate) S checkRate)  or
5.     (not(successPacer or failurePacer) S checkPacer)
6.   )
7. )

Validation handler:
{ reject_reg <= '1'; }
```

Figure 3.11: CheckProgrammerCommands Property

was easier to express with recovery being on validation.

As a final consideration, note that the handlers of SafeCounterModify, InterruptFix and NoDisableWhileConverting can be invoked simulteously if an incorrect value is written to cntr_cntrl2, which results in the execution of multiple bus writes. However, this causes no problem since all handlers overwrite cntr_cntrl2 with the same valid value.

3.5 Case Study: Pacemaker

We now describe a second case study relative to the pacemaker SoC implementation described in Section 2.2, to show how monitors can be directly exploited to enforce correct communication between lower and higher criticality applications. Recall that the Programmer process is used to update execution parameters of both the Pacer and Rate Adapter based on received RF commands. This is potentially dangerous, because both the Pacer and Rate Adapter are more critical than the Programmer and RF process. To solve the problem, we can introduce a commit protocol. The Programmer sends new parameters followed by a `check` command to the Rate Adapter through the *RateAdapter.parameters* message queue. The Rate Adapter validates the received parameters and sends back either a `success` or a `failure` answer to the *Programmer.response* message queue. The Programmer then repeats the same steps for the Pacer using its *Pacer.parameters* queue. If the Programmer receives `success` answers from both processes, it then sends `commit` messages to the Rate Adapter and Pacer process, causing them to load the received parameters. Unfortunately, since the Programmer is a complex, non safety-critical process, it could fail after sending a commit command to just one of the two processes. While this would not compromise the life-critical functionality (the Pacer control algorithm rejects any unsafe control points [10]), it can nevertheless disrupt the Rate Adapter causing significant discomfort to the patient.

The solution that we adopt is to send the `commit` command directly from the monitor; in this way, we isolate the critical functionality of the Programmer module inside the certified wrapper. A set of Programmer events are specified in the `EventList` in Figure 3.9, consisting of `check` and `commit` commands and `success` and `failure` answers for both the Pacer and the Rate Adapter. The *SendPacerCommit* property in Figure 3.10 is used to send the `commit` command to the Pacer on validation (an equivalent property with different handler can be used to send the `commit` to the Rate Adapter). In the PTLTL formula, `<*>`, `(*)` and `S` are temporal operators denoting *eventually in the past*, *previously* and *since*. Line 1 specifies that a `success` is received from the Pacer in the present and a `success` from the Rate Adapter has been received in the past. Line 2 implies that at least one `successRate` event has been received since the previous `successPacer` (if any), and Line 3 implies that at least one `successRate` has been received since the previous `failureRate` (if any); this makes sure that a valid parameter set has been passed to the Rate Adapter since the last commit operation. Finally, property *CheckProgrammerCommands* in Figure 3.11 is used to reject any erroneous Programmer command. Line 1 is used to reject `commit` commands from the Programmer, since only the monitor should send them. Lines 2-7 make sure that the Programmer can not send a `check` command if it has not received an answer (either a `success` or a `failure`) to its previous `check` commands to both the Pacer and Rate Adapter: otherwise, the Programmer could send a `check` command immediately before a `commit` is sent by the monitor, causing a wrong set of parameters to be loaded.

3.6 Related Work

There are two main run-time verification approaches: 1) offline, where a log, or trace is kept, which can then be used for purposes of debugging; and 2) online, where a property is checked while the program is running. As HardwareMOP is an online technique, we will only describe online approaches to runtime verification.

MaC [50], PathExplorer (PaX [42], and Eagle [12] use specific verification languages which cannot be changed, while HardwareMOP, as an extension of MO [22], will eventually support all the logics supported in JavaMOP. Temporal Rove [25] is a commercial runtime verification tool which uses future time metric temporal logic. It provides inline specification of monitors, where the monitors are written straight in the source file. Inline specification does not make sense for HardwareMOP, as there is no program being monitored per se. Program Query Language (PQL [62], is an approach somewhat similar to MOP, although it also only allows one specification language. PQL can support the full generality of context free languages. Tracematches [7] is very similar to JavaMOP. The biggest difference is that its choice of regular expressions for logical formalism is hardwired. It is an extension of the AB [6] AspectJ compiler. All of the above approaches are designed to monitor specific programs, and are implemented in software. This has the effect of both adding runtime overhead, and performing a function different from that of HardwareMOP, which monitors communication among hardware components.

The PSL to Verilog compiler, P2 [60], is the sole attempt to perform formal runtime verification in hardware, of which we are aware. P2V is similar to HardwareMOP in that monitors are implemented in hardware rather than

software, and that both approaches thus have no runtime overhead on the CPU. P2V, however, is more like the above approaches in that it is designed for monitoring actual programs rather than peripheral devices. Also it requires a dynamically extensible soft-core processor implemented on an FPGA, while our approach can potentially be applied to any COTS communication architecture. Further, P2V uses hardwired logic while HardwareMOP allows different formalisms.

Finally, in recent years there have been several proposals in the industry to extend virtual memory support to peripherals (for example, see [29]). While the main objective of these mechanisms is to extend virtualization to hardware devices, they can nevertheless be used to improved system reliability preventing peripherals from writing to wrong locations in main memory. In comparison, HardwareMOP is able to check a much greater range of requirements.

3.7 Future Work

There are several interesting directions in which our HardwareMOP framework could be further developed. First of all, HardwareMOP could be extended to support other logic specifications. Most of the plugins already developed for the MOP framework will require little work, with the exception of context free grammars (CFG)[2], which would require implementing, effectively, a hardware LR(1) parser. This extension is not trivial: the monitor must be able to process each event in few clock cycles, but a LR(1) parser can perform an unbounded number of reductions each event. More importantly, none of the formalisms supported so far include the notion of time as an explicit variable, making it difficult to check real-time requirements. As a long term research goal to solve this problem, we plan to develop efficient runtime monitoring algorithms for a significant subset of timed automata [4].

Second, the specifications of HardwareMOP properties could be better integrated in the system design flow. In particular, functional and logical requirements are entirely separated in the AADL-based specification for our SoC platform in Section 2.2.2. We believe that some formal properties could be automatically extracted from the AADL model without designer intervention, especially if more precise behavioral information is added to each process (for example, AADL has been recently extended with a behavioral annex); however, support for timed automata would be required to bridge the gap between the specification of functional and timing properties. In alternative, a complete set of acceptable run-time behavior could be possibly generated based on an executable functional model of the system, for example in a description language such as Metropolis [30]. Finally, currently the designer is responsible for manually checking that the set of specified functional properties are sufficient to ensure system correctness. To simply this step, we would like to integrate our tools with a model checking framework like Maude [31] or UPPAAL [101]; run-time properties should be directly exported to the model-checking tool, facilitating system-level verification.

[2]More precisely, MOP supports DCFLs.

Chapter 4

Memory Interference Analysis

The predictable integration of COTS components in real-time systems poses significant challenges from a timing perspective. These problems find their roots in the fact that computer industry is mainly driven by average performance paying very little attention to worst case analysis which is the "workhorse" of real-time theory and schedulability analysis. In particular, we focus on the interaction between the CPU cache and DMA peripherals contending for shared main memory access. Unfortunately, most COTS architectures feature a single-port main memory that is shared among all CPU cores and peripherals. When a task suffers a cache miss, contention for access to main memory can significantly delay cache line fetch and greatly increase the Worst Case Execution Time (WCET) of the task. We performed an experiment on a standard Intel dual core platform to understand the severity of this issue (details are provided in Section 4.4.4). We engineered a task that continuously suffers cache misses and measured its WCET while running it in isolation. We then added to the experiment a second copy of the task running on the other core and a PCIe [69] peripheral using DMA to saturate main memory with write requests, and measured a WCET increase of 196% for the task.

In this chapter, we introduce a novel WCET analysis framework that can model the interaction between CPU and peripherals contending for shared main memory and compute memory delay bounds for a task under analysis. In particular, we provide three main contributions: **(1)** we describe models to capture the profile of cache misses suffered by a given task over time, and discuss how to derive such information using either static analysis or experimental measurements. **(2)** We introduce the key idea of computing a memory traffic arrival curve for each core, given a set of executed tasks. The arrival curve provides an upper bound to the amount of memory traffic generated by the core in any interval of time. **(2)** We describe algorithms that compute delay bounds for a task given traffic curves for all other cores and peripheral buses in the system. Our framework is able to distinguish the behavior of DMA peripherals, whose traffic is buffered, from the behavior of CPU cores, which stall on cache misses. Furthermore, the obtained bound is tight in single core systems.

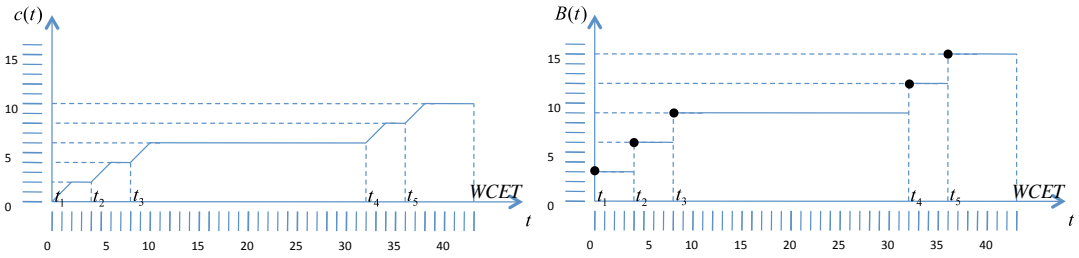
4.1 System Model

We consider a COTS system composed of multiples *processing cores* implemented on CPU dies. Each processing core PE_i can employ one or more cache levels, but caches are private and not shared with other cores. Cache misses in the last cache level generate requests to the shared main memory, which must arbitrate among simultaneous requests by different cores.

We assume that a specification for the arbitration scheme is available, which is usually the case for COTS components used in embedded platforms (see [63] for example). Different types of interconnection between CPU die and the rest of the system are available: alternatives include direct connection through a memory controller implemented on the CPU die, dedicated buses (Front Side Bus) implemented as a system chip (also known as *northbridge*) located on the motherboard, and switching architectures such as AMD HyperConnect. In general, we are not interested in covering each and every interconnection type: in almost the totality of systems, main memory is implemented as dynamic, single-port RAM, and the memory bus is the main data bottleneck in the system. Hence, in this work we focus our attention on the arbitration for access to the main memory bus, ignoring the details of the system interconnection. We shall make vary general assumptions, namely, arbitration can follow either a Round-Robin (RR), First-Come-First-Serve (FCFS), or Fixed Priority (FP) scheme.

Processing cores execute periodic tasks. The set of tasks executed on each core is static and tasks are not allowed to migrate among cores (partitioned scheduling). We do not assume any synchronization among schedules: each core can run asynchronously with respect to other cores. Scheduling follows a *restrictive preemption model*: the control flow graph for each task τ_i is divided into a series of S_i sequential *scheduling intervals* $\{s_{i,1}, \ldots, s_{i,S_i}\}$. Each scheduling interval can include branches and loops, but scheduling intervals must be executed in sequence. Multiple tasks executed on the same processing core can be scheduled according to either fixed time slots (cyclic executive), with a given set of scheduling intervals assigned to each slot, or according to a real-time on-line scheduling algorithm such as Earliest Deadline First (EDF) or Rate Monotonic (RM); however, preemption is only allowed at the end of the scheduling interval and not during its execution. This model allows us to bound the effect of preemptions on cache content: a preempting task could eject τ_i's instructions and data from cache, thus increasing the number of cache misses suffered by τ_i. Since preemption can only happen between time slots, we can determine the worst case number of cache misses in each scheduling interval by simply assuming that the state of the cache is unknown before the start of each time slot.

Each task τ_i is characterized by a *cache profile* c_i^{prof}, which encodes information on the pattern of cache misses suffered by the task. We consider two different models for the cache profile, which provide a tradeoff in representation accuracy versus complexity of deriving the cache profile. In the *exact model*, we assume that the task suffers a total of N cache misses, and that the time at which each cache miss happen is known exactly. In particular, we denote the set of N misses, in temporal order, as $\{f_1, f_2, \ldots, f_N\}$. The cache profile is comprised by a set of miss start times $\{t_1, \ldots, t_N\}$ and a set of miss lengths $\{L_1, \ldots, L_N\}$: for each miss f_j, t_j is the time at which the miss happens,



(a) *Access Function* $c(t)$. **(b) *Blocking Function* $B(t)$.**

Figure 4.1: Cache Functions.

triggering a cache line fetch, and L_j is the time required for the fetch operation, assuming no interference in memory from peripherals or other cores. Note that in a write-back cache, whenever a dirty cache line must be replaced, the cache controller generates two requests to main memory: a write for the replaced cache line, and a read for the fetched cache line. In this case, L_j includes the time required for both operations. A useful representation of the cache profile c_i^{prof} for a given task is provided by the *cache access function* $c(t)$. An example of cache access function is plotted in Figure 4.1(a) for a profile with $N = 5$ fetches and $L_1, \ldots, L_5 = 2ms$. The x axis represents time between 0 and the WCET of the task unmodified by peripheral interference, while the y axis represents the cumulative cache fetch time in memory. A slope of 1 indicates a fetch operation, while a slope of 0 indicates that the CPU is executing task code.

An exact cache profile can be obtain through testing. While testing can fail to reveal the real worst case timing for the task, it is nevertheless a commonly used procedure. In general, assume that task τ_i is tested with a set of M_i different test vectors. Then M_i cache profiles $c_{i,1}^{prof}, \ldots, c_{i,M_i}^{prof}$ can be experimentally derived by measuring both execution time and all cache misses; this is typically only achievable by running the task in a cycle-accurate CPU emulator. Note that it is necessary to keep track of multiple cache access functions since some experiments can account for a longer worst case computation time but smaller amount of cache misses, and therefore less potential delay, or vice versa.

To simplify the derivation of cache profiles and avoid considering multiple profiles for each task, we consider a second, *interval-based model*. In the interval-based model, each cache profile is provided as a tuple: $c_i^{prof} = \{exec_{i,j}^L, exec_{i,j}^U, \mu_{i,j}^{\min}, \mu_{i,j}^{\max}\}$. $exec_{i,j}^L$ and $exec_{i,j}^U$ are lower and upper bounds on computation times for scheduling interval $s_{i,j}$ assuming that memory operations take zero time, e.g. they are the times required to execute the instructions in the scheduling interval[1]. $\mu_{i,j}^{\min}, \mu_{i,j}^{\max}$ are the minimum and maximum number of access requests to main memory in scheduling interval $s_{i,j}$. The way $\mu_{i,j}^{\min}, \mu_{i,j}^{\max}$ are computed depends on cache architecture. In a write-back cache, both requests to write-back the replaced cache line and to read the fetched cache line must be accounted for in $\mu_{i,j}^{\min}, \mu_{i,j}^{\max}$.

For the interval-based model, maximum and minimum scheduling interval time and number of memory requests

[1]We instead use the symbol e to refer to the execution time for a scheduling interval, e.g. the time required to complete the interval, inclusive of memory operations.

can be obtained by either experimental measures or static analysis. Using static analysis, it is possible to obtain two worst-case execution traces for each interval: one which maximizes computation time, and a second one that maximizes number of cache misses. The same holds for best-case traces. Details on our experimental measurement methodology are provided in Section 4.4.3. In this case, multiple execution traces must also be derived; note that the required computation times can be obtained from the measured task execution times by subtracting the time required for each catch fetch/replacement, which is assumed to be constant when the task is not subject to interference in main memory. To reduce all traces to a single cache profile, we consider the maximum computation time among all traces and the maximum number of memory requests among all traces, independently from each other (the same holds for minima). Finally, note that both cache profile models implicitly assume that if the CPU fetch unit is delayed Δ time units in a scheduling interval, its execution time increases by at most Δ. Modern CPU architectures can exhibit timing anomalies, in the sense that it is possible to produce a trace where the worst case computation time is produced when a specific memory access results in a cache hit rather than a miss; this is because the state of the pipeline depends on the time required for each memory access. Therefore, we assume a CPU architecture where execution time and communication time can be decoupled [105]. If that is not possible, then the (pessimistic) bounds computed by timing analysis must capture all effects of timing anomalies. In particular, the worst case execution time for a trace must be computed considering the uncertainties in the pipeline state due the fact that each memory access can result in either a cache hit or miss. Taking this uncertainty into consideration ensures that given a bound $exec^U$ on the computation time (without memory operation time) for a trace, if the total time required for all memory operations is no more than Δ, then $exec^U + \Delta$ is a valid (pessimistic) bound on the whole execution time for the trace.

Finally, we assume that peripheral traffic can be injected into main memory. In a typical COTS system, peripherals are connected through a dedicated interconnection such as the Peripheral Component Interconnect (PCI) and related standards (PCI-X and PCI Express). We assume that each peripheral in the system is characterized by an upper arrival curve $\alpha^*(t)$: for each interval of time t, $\alpha^*(t)$ represent the maximum amount of time required by the peripheral in main memory to perform DMA operations. Note that before reaching main memory, peripheral requests are buffered in interconnection elements such as bridges (PCI/PCI-X) and switches (PCI Express): as such, all peripheral requests coming from the same interconnection must be aggregated into a single *buffered flow* representing the cumulative requests produced by a given interconnection on main memory. An analysis to aggregate peripheral traffic on the PCI bus is presented in [65].

To simplify exposition, in the rest of this chapter we first show analysis results for simpler models with very restrictive assumptions, and then progressively remove such assumptions to analyze more complex and realistic systems. In Section 4.2, we first show how to compute an upper delay bound for a single task running on a single core system due to peripheral interference in main memory, assuming the exact model for its cache profile. We employ a set of simplifying assumptions regarding both memory arbitration and peripheral flows. Under such assumptions, we show that the obtained delay bound is tight, in the sense that there exists a pattern of peripheral accesses to main memory that

results in the computed delay. Then, in Section 4.3 we show how to extent the analysis to multiple tasks scheduled on a single core. Finally, in Section 4.4 we show how to compute delay bounds for tasks running on a multicore system, due to interference from both peripherals and other cores. Furthermore, we consider the more realistic interval-based model for cache profile and make very general assumptions on main memory arbitration and cache. The downside to this more general model is that the derived delay bound is no longer tight; however, we argue and show experimental evidences that in practice the bound is close to the real worst-case.

4.2 Single Task Analysis

In this section, we focus on the derivation of delay bounds for a single task τ_i running on a single core system. The exact model is assumed for the cache profile of the task. Without loss of generality and to simplify notation, in the remaining of this section we drop the subscript i of the task under analysis and use c^{prof} for its cache profile. We assume that whenever the CPU suffers a cache miss, it fetches a single cache line taking a constant time $L_j = L$ to complete the operation; hence, given the cache profile $\{f_1, f_2, \ldots, f_N\}$, we use both the term cache miss and cache fetch to refer to each f_j, interchangeably. The arbitration of the main memory bus is supposed to follow a round-robin scheme.

We assume that all peripherals are directly connected to memory without buffering. In this case, the arrival curves for all peripherals can be aggregated into a single flow, which we represent with a load bound function $E(t)$: in any interval of length t, $E(t)$ represents the amount of time required by all peripherals in main memory. Furthermore, let L' be the maximum length in time of any non-preemptive peripheral transaction. Then the following properties for $E(t)$ follow by definition.

Property 1 *$E(t)$ is monotonically non decreasing.*

Property 2 *The maximum slope of $E(t)$ is equal to 1, i.e. $\forall t', t'', 0 \leq t' \leq t'' : E(t'') - E(t') \leq t'' - t'$.*

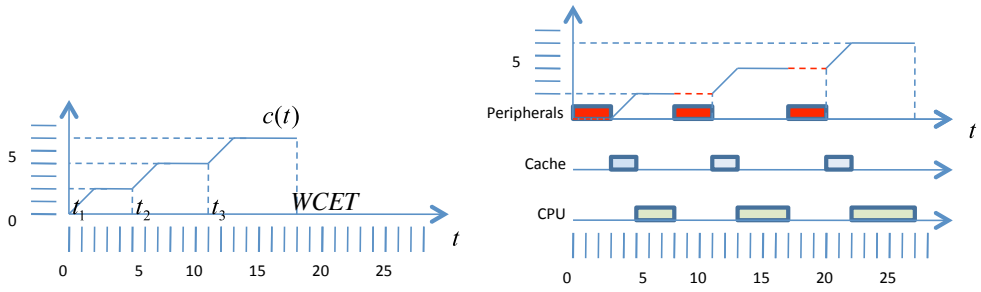
Property 3 *$E(t)$ is continuous.*

Property 4 $\forall t, 0 \leq t \leq L' : E(t) = t$

Property 5 $\forall t', t'' \geq 0 : E(t') + E(t'') \geq E(t' + t'')$

Proof.

Property 2 follows from the fact that in any interval t', t'' the memory bus can not service peripheral transactions for more than $t'' - t'$. Property 3 then follows from Properties 1, 2. Since in the upper bound case a transaction of up to length L' can be executed in interval $[0, L']$, Property 4 follows. To prove Property 5, it is enough to consider that any



(a) ***Cache Access Function.*** **(b)** ***Modified Schedule.***

Figure 4.2: Example of Worst Case Delay.

interval of length $t' + t''$ can be decomposed in two consecutive intervals of length t' and t''; since E(t) is an upper bound to the requested time in any interval, it can never be the case that $E(t') + E(t'') \leq E(t' + t'')$. □

Furthermore, to keep the analysis interesting we shall also assume the following property, which simply states that a peripheral can not continuously occupy the bus.

Property 6 *$E(t)$ is not the identity function.*

The goal of the analysis is to compute the worst case delay that the task can suffer due to peripheral activity. In particular, for all $t', t'' : 0 \leq t' \leq t'' \leq WCET$, we denote with $D(t', t'')$ the cumulative delay suffered by all cache misses in the cache profile c^{prof} with start time t_i within the closed interval $[t', t'']$. As a consequence, the overall task delay is equal to $D(t_1, t_N)$. The analysis is based on the following main idea: we first compute an upper bound Ub to the maximum delay $D(t_i, t_j)$ for all i, j, meaning $D(t_i, t_j) \leq Ub$. We then progressively decrease the bound by taking the intersection of multiple such constraints. The round robin arbitration assumption leads to the following property, which will be used in later proofs:

Property 7 *Under round robin arbitration, each cache fetch operation can be delayed by at most one full peripheral transaction of length L', assuming that the peripheral transaction is requested at the same time as the fetch request and the last previously completed bus operation was a fetch.*

Using the round robin assumption, we will prove that the final analysis bound is *tight*, meaning that there exists a pattern of peripheral accesses to the memory bus consistent with $E(t)$ that results in $D(t_i, t_j)$ being equal to the computed bound. The definition of a consistent transaction schedule is provided below.

Definition 3 *A schedule of peripheral transactions on the memory bus is said to be consistent with upper bound $E(t)$ if and only if there exists L' such that every transaction has length at most L' and for each interval $[t', t'']$, the time the bus is occupied executing peripheral transactions in $[t', t'']$ is at most equal to $E(t'' - t')$.*

An example of worst case interaction between the cache controller and master peripherals is shown in Figure 4.2, where $L' = 3, L = 2$ms. In Figure 4.2(a), the cache function $c(t)$ for the original profile c^{prof} is represented. In Figure 4.2(b), a new bus schedule is represented where a peripheral transaction of length L' is requested each time the cache controller is starting fetch f_i following Property 7. We also plot the modified cache access function when peripheral transactions are taken into account; note that the starting time of each fetch f_i in the modified profile is now equal to $t_i + D(t_1, t_i)$. We can capture this worst case scenario by introducing a new *cache blocking function* $B(t)$ as follows:

$$\begin{cases} \forall t, t < t_1 : & B(t) = 0 \\ \forall i, 1 \leq i < N, \forall t, t_i \leq t < t_{i+1} : & B(t) = iL' \\ \forall t, t \geq t_N : & B(t) = NL' \end{cases} \tag{4.1}$$

The $B(t)$ function associated with the cache access function $c(t)$ of Figure 4.1(a) is shown in Figure 4.1(b). The following lemma formally proves that $B(t)$ represents an upper bound on the cumulative time that fetch operations can be delayed.

Lemma 4 *Assuming round robin arbitration for the memory bus, for each $i, j, i \leq j : B(t_j) - B(t_i^-)$ is an upper bound for $D(t_i, t_j)$, where $B(t^-) = \lim_{x \to t^-} B(x)$ (by definition, $B(0^-) = 0$). Furthermore, there exists $E(t)$ such that the bound is tight.*

Proof.

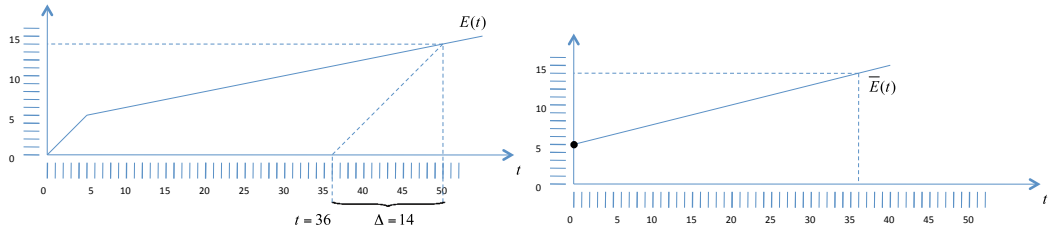
Since the number of cache fetches in $[t_i, t_j]$ is equal to $(j-i+1)$, from Property 7 it follows: $D(t_i, t_j) \leq (j-i+1)L'$. Furthermore, since $B(t_j) - B(t_i^-) = (j - i + 1)L'$ by definition, we obtain that $D(t_i, t_j) \leq B(t_j) - B(t_i^-)$. Finally, assume that $\forall t, 0 \leq t \leq t_j - t_i + (j - i + 1)L' : E(t) = t$ (i.e. the bus can be always busy in the interval under analysis), one peripheral transaction of length L' is requested at the same time of each cache fetch request, and initially high rotating (round robin) priority is assigned to peripherals at time 0. Then it is easy to see (refer to Figure 4.2) that $D(t_i, t_j) = (j - i + 1)L'$, concluding the proof. □

In general, the bound expressed by Lemma 4 is not tight, since $E(t)$ can constrain the peripheral activity to be less than $B(t_j) - B(t_i^-)$. Hence, to obtain a tight bound it is also necessary to express a bound based on $E(t)$.

Lemma 5 *Let $\bar{E}(t) = \max\{\Delta | \Delta = E(t + \Delta)\}$. Then $\bar{E}(t_j - t_i)$ is an upper bound to $D(t_i, t_j)$.*

Proof.

Assume that the cumulative delay for fetches $\{f_i, \ldots, f_j\}$ is Δ. It follows that in an interval of length $t_j - t_i + \Delta$, an amount of peripheral transactions at least equal to Δ must have been scheduled on the memory bus, hence it must hold that $\Delta \leq E(t_j - t_i + \Delta)$. Therefore, by definition $\Delta \leq \bar{E}(t_j - t_i)$, which implies that $\sup\{\Delta | \Delta \leq E(t + \Delta)\}$ is an upper bound to $D(t_i, t_j)$.



(a) ***Upper Bound Function*** $E(t)$**.** **(b)** ***Modified Upper Bound Function*** $\bar{E}(t)$**.**

Figure 4.3: Peripheral Load Functions.

It remains to show that $\sup\{\Delta|\Delta \leq E(t+\Delta)\} = \max\{\Delta|\Delta = E(t+\Delta)\}$. First note that the supremum can not be infinite, since this would imply that $E(t)$ has an asymptote with slope equal to 1 (note that $E(t)$ can never have slope strictly greater than 1 according to Property 2). However, unless $E(t)$ is the identity function, which is not allowed according to Property 6, it is easy to see that an asymptote with slope 1 would violate Property 5. Finally, since $E(t)$ is continuous according to Property 3, at the supremum it must hold: $\Delta = E(t+\Delta)$, which implies $\sup\{\Delta|\Delta \leq E(t+\Delta)\} = \max\{\Delta|\Delta = E(t+\Delta)\}$. □

A graphical representation of $\bar{E}(t)$ is shown in Figure 4.3(b) based on a given $E(t)$ function. Intuitively, $\bar{E}(t)$ is the ordinate of the intersection of $E(x)$ with the line $x - t$ (where t is fixed and x varies). An example application of Lemma 5 is shown in Figure 4.4 using the functions of Figures 4.1,4.3. Following Lemma 4, a bound for $D(t_1, t_3)$ is $B(t_3) = B(8) = 3L' = 9$. However, the bound is not tight: since $\bar{E}(8) = 7$, it follows that the cumulative delay of fetches $\{f_1, f_2, f_3\}$ can be no more than 7. Intuitively, this means that although the cache controller performs 3 fetches in the interval, the maximum delay of 9 ms can never be attained since the load imposed by peripheral activity is not high enough.

Note that in Lemma 5, t_i and t_j represent the original start times of f_i, f_j in the unmodified schedule, that is, they do not shift to take into account the delay incurred by each fetch, which might seem counterintuitive. The main intuition behind this result is that there is a circular dependency between the amount of peripheral load that interferes with $\{f_i, \ldots, f_j\}$ and the delay $D(t_i, t_j)$: when peripheral traffic is injected on the memory bus, the start time of each fetch is delayed. In turn, this increases the time interval between f_i and f_j and therefore more peripheral traffic can now interfere with those fetches. Our key idea is that we do not need to modify the start times $\{t_i, \ldots, t_j\}$ of fetches when we take into account the peripheral traffic injected on the memory bus: instead, we can take this effect into account using the equation that defines $\bar{E}(t)$, where Δ represents both the maximum delay suffered by fetches and the increase in the time interval for interfering traffic.

Since Lemmas 4,5 express bounds on the same interval, they can be trivially combined yielding the following result:

Theorem 6 *For each* $i, j : i \leq j$, $\min(B(t_j) - B(t_i^-), \bar{E}(t_j - t_i))$ *is an upper bound for* $D(t_i, t_j)$.

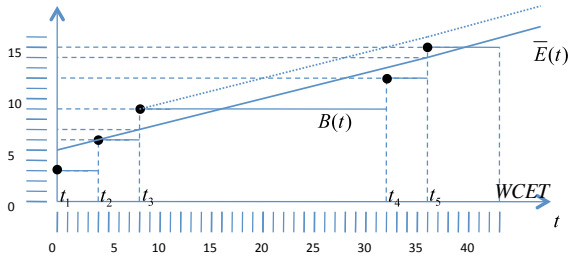


Figure 4.4: Example Bound on Peripheral Load.

Unfortunately, the bound expressed by Theorem 6 is not tight. To understand why, refer again to Figure 4.4. If we apply Theorem 6 to fetches $\{f_1, \ldots, f_5\}$, we obtain $D(t_1, t_5) \leq \min(15, 14) = 14$. However, note that since $D(t_i, t_j)$ is an upper bound, it must hold $D(t_1, t_5) \leq D(t_1, t_3) + D(t_4, t_5)$ (similarly to Property 5 for $E(t)$). By applying Theorem 6 again we obtain $D(t_1, t_3) \leq \min(9, 7), D(t_4, t_5) \leq \min(6, 6)$, hence $D(t_1, t_5) \leq 13$, which is less than the bound of 14 previously computed. The example shows that in order to compute a tight bound on $D(t_i, t_j)$, multiple subintervals may need to be analyzed. The following lemma shows how to take bounds on subintervals into account.

Lemma 7 *Let $i \leq j < k$, and Ub be an upper bound to $D(t_i, t_j)$. Then $D(t_i, t_k) \leq Ub + \min\big(B(t_k) - B(t_{j+1}^-), \bar{E}(t_k - t_i) - Ub\big)$.*

Proof.

Applying Lemma 4, $D(t_{j+1}, t_k) \leq B(t_k) - B(t_{j+1}^-)$. Since furthermore $D(t_i, t_k) \leq D(t_i, t_j) + D(t_{j+1}, t_k)$, it follows that $D(t_i, t_k) \leq Ub + B(t_k) - B(t_{j+1}^-)$. By applying Lemma 5, $D(t_i, t_k) \leq \bar{E}(t_k - t_i)$. Combining the two bounds yields the lemma. □

Lemma 7 has an interesting graphical intuition. Consider again the example of Figure 4.4, using a bound $Ub = 7$ for $D(t_1, t_3)$. To determine what is the maximum delay contribution of fetches $\{f_4, f_5\}$ to $D(t_1, t_5)$, we can simply "raise" $\bar{E}(t)$ by a value equal to $B(t_3) - Ub = 2$ and then check if the function intersects $B(t)$ at $t = t_5$ to determine which of the two bounds in the minimization function is more restrictive (the raised function is dotted in Figure 4.4).

Based on the intuition of Lemma 7, we now introduce main Algorithm 1. It iteratively computes a delay term u_k for each fetch f_k and sums the delay terms together to obtain a bound $Ub(t_1, t_k) = \sum_{j=1}^{k} u_j$ for $D(t_1, t_k)$; $Ub(t_1, t_k)$ is then used to compute a new bound for $D(t_1, t_{k+1})$ and therefore a delay term u_{k+1}, and so forth until the final bound $Ub = Ub(t_1, t_N)$ for $D(t_1, t_N)$ is obtained. In the algorithm description, Q is a list of pairs (t_i, y_i) ordered by increasing values of t_i. For each pair, t_i is the unmodified start time of fetch f_i, while y_i represents the accumulated delay: at step k of the algorithm, $y_i = \sum_{j=i}^{k-1} u_j$, i.e. y_i is a bound $Ub(t_i, t_{k-1})$ to the delay $D(t_i, t_{k-1})$.

We now prove that Algorithm 1 computes a tight upper bound Ub to $D(t_1, t_N)$. For ease of exposition, we split the proof in two. Theorem 8 proves that Ub is a valid upper bound to $D(t_1, t_N)$. Theorem 10 then shows that there

Algorithm 1 Compute $D(t_1, t_N)$

1: $Ub := 0$
2: $Q := \{\}$
3: **for** $k = 1 \dots N$ **do**
4: add $(t_k, 0)$ to Q
5: compute $u_k = \min\left(B(t_k) - B(t_k^-), \min_{(t_i,y_i)\in Q}\{\bar{E}(t_k - t_i) - y_i\}\right)$
6: **for all** (t_i, y_i) in Q **do**
7: $y_i := y_i + u_k$
8: **end for**
9: $Ub := Ub + u_k$
10: **end for**
11: **return** Ub

exists a feasible schedule of peripheral transactions that results in $D(t_1, t_N) = Ub$, hence the bound is tight.

Theorem 8 *Algorithm 1 computes an upper bound* Ub *to* $D(t_1, t_N)$.

Proof.

The proof proceeds by induction on k, proving the following property: $\forall k \geq 1, Ub(t_1, t_k)$ is an upper bound to $D(t_1, t_k)$.

Base Case: We need to prove that $Ub(t_1, t_1)$ is an upper bound to $D(t_1, t_1)$. Following the algorithm, $Ub(t_1, t_1) = u_1 = \min\left(B(t_1) - B(t_1^-), \bar{E}(t_1 - t_1) - y_1\right)$ with $t_1 = t_1, y_1 = 0$. Therefore, $Ub(t_1, t_1)$ is equal to the bound computed by Theorem 6 for $D(t_1, t_1)$, concluding the proof obligation.

Induction Step: Assume that $\forall j < k, Ub(t_1, t_j)$ is an upper bound to $D(t_1, t_j)$. We need to prove that $Ub(t_1, t_k)$ is an upper bound to $D(t_1, t_k)$. By contradiction, assume that $Ub(t_1, t_k)$ is not an upper bound to $D(t_1, t_k)$. Then since $Ub(t_1, t_k) = Ub(t_1, t_{k-1}) + u_k$, it follows that at least one of the following two assertions is true for any schedule that produces a delay $D(t_1, t_k)$: the delay suffered by $\{f_1, \dots, f_{k-1}\}$ is strictly greater than $Ub(t_1, t_{k-1})$; or the delay suffered by f_k is strictly greater than u_k. However, the first assertion is impossible due to the induction hypothesis. Hence, let the delay suffered by f_k be $u_k + \Delta$, with $\Delta > 0$. We consider two cases relative to which of the two arguments of the minimization function in line 5 is returned.

- $u_k = B(t_k) - B(t_k^-)$: then following Lemma 4 applied to $D(t_k, t_k)$, if follows $\Delta = 0$, a contradiction.
- Let i be the index of the pair (t_i, y_i) which yields the minimum. We show that the delay for $\{f_1, \dots, f_{k-1}\}$ is at most equal to $Ub(t_1, t_{k-1}) - \Delta$, which contradicts the fact that $Ub(t_1, t_k)$ is not an upper bound to $D(t_1, t_k)$. By definition of Algorithm 1, it follows $y_i = Ub(t_i, t_{k-1})$, and therefore $u_k = \bar{E}(t_k - t_i) - Ub(t_i, t_{k-1})$, or equivalently $\bar{E}(t_k - t_i) = Ub(t_i, t_k)$. Since according to Lemma 5 $D(t_i, t_k) \leq \bar{E}(t_k - t_i)$, it follows that if the

delay suffered by f_k is $u_k + \Delta$, then the delay suffered by $\{f_i, \ldots, f_{k-1}\}$ is at most equal to $Ub(t_i, t_{k-1}) - \Delta$. To conclude the proof, it now suffices to note that the delay suffered by the remaining fetches $\{f_1, \ldots, f_{i-1}\}$ can not be greater than $Ub(t_1, t_{i-1})$ due to the induction hypothesis.

□

Lemma 9 *Consider a schedule of peripheral transactions where a single transaction of length at most L' is requested at the same time of each cache fetch, and assume that for each i, j the time the bus is occupied executing peripheral transactions that delay fetches $\{f_i, \ldots, f_j\}$ is at most equal to $\bar{E}(t_j - t_i)$. Then the schedule is consistent with upper bound $E(t)$.*

Proof.

According to Definition 1, we need to prove that for each t', t'', the time the bus is occupied executing peripheral transactions in $[t', t'']$ is at most $E(t'' - t')$. Following Property 2, it suffices to check all instants t' corresponding to the beginning of a peripheral transaction and all instants t'' corresponding to the end of a peripheral transaction. Therefore, consider any set of fetches $\{f_i, \ldots, f_j\}$ and the corresponding peripheral transactions, and let Δ' be the total time the bus is occupied executing those transactions. The total time distance between the beginning of the first transaction and the end of the last transaction is equal to $t_j - t_i + \Delta'$ since $D(t_i, t_j) = \Delta'$. By hypothesis, $\Delta' \leq \bar{E}(t_j - t_i)$. Due to Properties 2, 3 and since $\bar{E}(t_j - t_i) = \sup\{\Delta | \Delta \leq E(t_j - t_i + \Delta)\}$, it must also hold: $\Delta' \leq E(t_j - t_i + \Delta')$, which concludes the proof. □

Theorem 10 *Algorithm 1 computes a tight upper bound Ub to $D(t_1, t_N)$.*

Proof.

According to Theorem 8, Ub is an upper bound. Therefore, to prove that Ub is tight we only need to show a feasible schedule of peripheral transactions that results in $D(t_1, t_N) = Ub$. As in Theorem 8, let $Ub(t_i, t_k) = \sum_{j=i}^{k} u_j$.

We construct the schedule as follow: we request a transaction of length u_k at the same time of each fetch f_k. Since $u_k \leq L'$ due to the $B(t_k) - B(t_k^-)$ bound on line 5 and applying Lemma 9, it remains to prove that for each $i, j : Ub(t_i, t_j) \leq \bar{E}(t_j - t_i)$. Following the same reasoning as in Theorem 8, at each step k and for each pair (t_i, y_i) in Q, $y_i = Ub(t_i, t_{k-1})$. Hence, line 5 implies the condition $u_k \leq \bar{E}(t_k - t_i) - Ub(t_i, t_{k-1})$ or equivalently $Ub(t_i, t_k) \leq \bar{E}(t_k - t_i)$. Since a pair (t_i, y_i) is inserted for each fetch f_i and the condition is checked against each successive fetch f_j, the theorem follows. □

The complexity of Algorithm 1 depends on the complexity of computing $\bar{E}(t)$, which we assume $O(K)$ for some parameter K; for example, if $\bar{E}(t)$ is determined through measurement, K depends on the number of measured

transactions. Algorithm 1 computes $\bar{E}(t)$ exactly $N(N+1)/2$ times, hence it has complexity $O(N^2K)$. Luckily, the run-time overhead of the algorithm can be significantly decreased if additional assumptions are made on $\bar{E}(t)$. The idea is to introduce suitable bounds on $\bar{E}(t)$ that let us prune some of the (t_i, y_i) pairs in Q. Specifically, we assume that there exist concave upper and lower bounds to $\bar{E}(t)$ that have constant slope after an initial interval. Formally, let $\bar{E}^l_{conc}(t)$ be a concave lower bound to $\bar{E}(t)$; in particular, if $E(t)$ itself is concave, then Lemma 11, shown below, proves that $\bar{E}(t)$ is concave as well, therefore we can set $\bar{E}^l_{conc}(t) = \bar{E}(t)$. Furthermore, let $\sigma > 0$ be such that $\bar{E}^l_{conc}(t) + \sigma$ is a concave upper bound to $\bar{E}(t)$. Finally, we assume that there exists time T and slope ρ such that $\bar{E}^l_{conc}(t)$ has constant slope after T, i.e. $\forall t \geq T : \bar{E}^l_{conc}(t) = \rho(t-T) + \bar{E}^l_{conc}(T)$. If there is no such T, we simply define $T = \infty$. Algorithm 2 then shows a modified version of Algorithm 1 with two pruning conditions. Intuitively, we can prune a pair (t_i, y_i) at step k if we are sure that $u_{k+1}, \ldots, u_N$ will not be lower bounded by the pruned pair.

Algorithm 2 Compute $D(t_1, t_N)$

$Ub' := 0$
$Q := \{\}$
for $k = 1 \ldots N$ **do**
    add $(t_k, 0)$ to Q
    compute $u_k = \min\left(B(t_k) - B(t_k^-), \min_{(t_i,y_i)\in Q}\{\bar{E}(t_k - t_i) - y_i\}\right)$
    for all (t_i, y_i) in Q **do**
        if $\exists (t_q, y_q)$ in Q: $t_q < t_i \wedge \bar{E}^l_{conc}(t_k - t_i) - y_i \geq \bar{E}^l_{conc}(t_k - t_q) - y_q + \sigma$ **then**
            remove (t_i, y_i) from Q
        end if
        if $\exists (t_q, y_q)$ in Q: $t_q > t_i \wedge t_k - t_q \geq T \wedge \bar{E}^l_{conc}(t_k - t_i) - y_i \geq \bar{E}^l_{conc}(t_k - t_q) - y_q + \sigma$ **then**
            remove (t_i, y_i) from Q
        end if
    end for
    for all (t_i, y_i) in Q **do**
        $y_i := y_i + u_k$
    end for
    $Ub' := Ub' + u_k$
end for
return Ub

Lemma 11 *If the load function $E(t)$ is concave, then the modified load function $\bar{E}(t)$ is also concave.*

Proof.

First note that by definition it holds:

$$\bar{E}(t) = \max\{\Delta | \Delta = E(t + \Delta)\} = E(t + \bar{E}(t))\}. \tag{4.2}$$

By contradiction, assume that $\bar{E}(t)$ is not concave. Then there must exist three points $\bar{x}_1 < \bar{x}_2 < \bar{x}_3$ such that:

$$\frac{\bar{E}(\bar{x}_2) - \bar{E}(\bar{x}_1)}{\bar{x}_2 - \bar{x}_1} < \frac{\bar{E}(\bar{x}_3) - \bar{E}(\bar{x}_1)}{\bar{x}_3 - \bar{x}_1}. \tag{4.3}$$

Consider now three new points x_1, x_2, x_3, with $x_i = \bar{x}_i + \bar{E}(\bar{x}_i)$ for $i = 1, 2, 3$. Then from Equation 4.2 it directly follows: $E(x_i) = \bar{E}(\bar{x}_i)$, and therefore $\bar{x}_i = x_i - E(x_i)$. We can now rewrite Equation 4.3 as follows:

$$\frac{E(x_2) - E(x_1)}{x_2 - x_1 - (E(x_2) - E(x_1))} < \frac{E(x_3) - E(x_1)}{x_3 - x_1 - (E(x_3) - E(x_1))}. \tag{4.4}$$

Now note that by Property 2 it must hold $\frac{x_2 - x_1}{E(x_2) - E(x_1)} > 1$, $\frac{x_3 - x_1}{E(x_3) - E(x_1)} > 1$, therefore we obtain:

$$\frac{E(x_2) - E(x_1)}{x_2 - x_1} < \frac{E(x_3) - E(x_1)}{x_3 - x_1}. \tag{4.5}$$

This implies that $E(t)$ is not concave, a contradiction. □

Theorem 12 *Algorithm 2 computes a tight upper bound Ub' to $D(t_1, t_N)$.*

Proof.

According to Theorem 10, Algorithm 1 computes a tight upper bound Ub to $D(t_1, t_N)$. Hence, it suffices to show $Ub' = Ub$. We prove that if a pair (t_i, y_i) is removed from Q at step k in Algorithm 2 due to pair (t_q, y_q), then $\forall l \geq k$, at step l in Algorithm 1: $\bar{E}(t_l - t_i) - y_i \geq \bar{E}(t_l - t_q) - y_q$. Hence, the u_l computed by Algorithm 2 is equal to the one computed by Algorithm 1, and therefore $Ub' = Ub$. We separately analyze the two prune conditions.

- Line 7: Since $\bar{E}^l_{conc}(t)$ is concave and furthemore $t_q < t_i$, it must hold $\bar{E}^l_{conc}(t_l - t_i) - \bar{E}^l_{conc}(t_k - t_i) \geq \bar{E}^l_{conc}(t_l - t_q) - \bar{E}^l_{conc}(t_k - t_q)$. Also, between steps k and l, y_i, y_q are incremented by the same amount $Ub(t_k, t_{l-1})$. Hence, since $\bar{E}^l_{conc}(t_k - t_i) - y_i \geq \bar{E}^l_{conc}(t_k - t_q) - y_q + \sigma$ at step k, it also holds $\bar{E}^l_{conc}(t_l - t_i) - y_i \geq \bar{E}^l_{conc}(t_l - t_q) - y_q + \sigma$ at step l. But since $\bar{E}(t_l - t_i) \geq \bar{E}^l_{conc}(t_l - t_i)$ and $\bar{E}(t_l - t_q) \leq \bar{E}^l_{conc}(t_l - t_q) + \sigma$, it follows $\bar{E}(t_l - t_i) - y_i \geq \bar{E}(t_l - t_q) - y_q$.

- Line 10: Since $t_k - t_q \geq T$ and $t_q > t_i$, it also holds $t_k - t_i \geq T$. Hence, $\bar{E}^l_{conc}(t_l - t_i) - \bar{E}^l_{conc}(t_k - t_i) = \rho(t_l - t_k) = \bar{E}^l_{conc}(t_l - t_q) - \bar{E}^l_{conc}(t_k - t_q)$. Following the same reasoning used for the previous pruning condition we again obtain $\bar{E}(t_l - t_i) - y_i \geq \bar{E}(t_l - t_q) - y_q$.

□

The pruning condition of line 7 can be checked efficiently by implementing Q as a doubly linked list ordered based on increasing values of t_i. It is then sufficient to move forward through the list only once at each step k computing

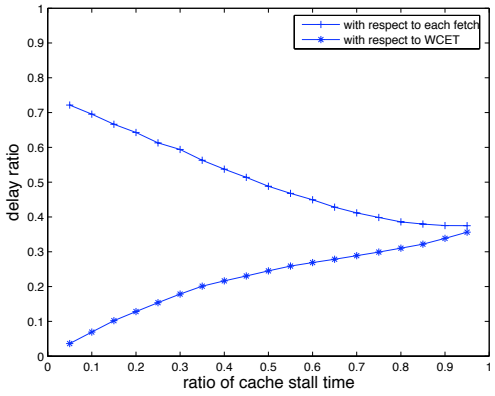


Figure 4.5: Delay ratios, 30% peripheral load.

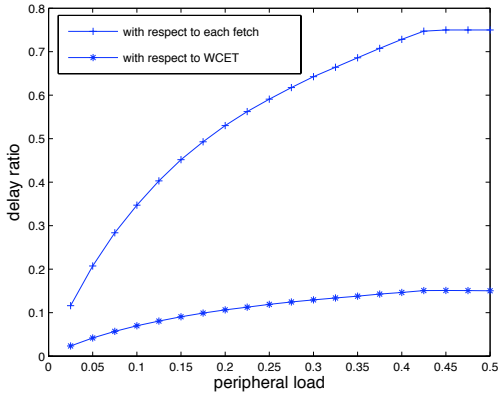


Figure 4.6: Delay ratios, 20% cache stall.

$\bar{E}^l_{conc}(t_k - t_i) - y_i$ for each pair (t_i, y_i) and keeping track of the minimum among such values. Similarly, the pruning condition of line 10 can be checked by moving backward through the list. As a consequence, the computational complexity of Algorithm 2 is still $O(N^2K)$. In Section 4.2.1 we show a synthetic evaluation of the relative computation times of Algorithms 1, 2.

Finally, note that while we formulated Algorithms 1, 2 to compute the overall task delay $D(t_1, t_N)$, the algorithms can be easily extended to compute a delay bound $D(t', t'')$ for any interval $[t', t'']$. Let $\{f_i, \ldots, f_j\}$ be the set of cache misses in profile c^{prof} with start times within $[t', t'']$; then it suffices to modify line 3 of the algorithms iterating k over $i \ldots j$ instead of $1 \ldots N$. The extension is used in Section 4.3.

4.2.1 Simulation Results

Synthetic simulations were performed to understand how the delay bounds computed by Algorithm 1 vary as a function of task parameters, and how much the optimization of Algorithm 2 can reduce the analysis computation time compared to Algorithm 1. We derived architecture parameters from publicly available data for a PowerPC architecture especially designed for embedded systems [63]; in particular, we used round robin arbitration, a cache line size of 32 bytes and a maximum non-preemptive peripheral transaction size of 24 bytes, which yields a L/L' ratio of $3/4$. We then generated synthetic bounds on peripheral load and cache access functions and computed the overall task delay $D(t_1, t_N)$ according to Algorithm 1. To obtain $E(t)$, we first generated sequences of 1000 peripheral transactions randomizing the interarrival time between transactions to obtain a desired value for the average peripheral load on the memory bus; we then analyzed all possible intervals to compute $E(t)$. Each generated cache profile c^{prof} is comprised of $N = 100$ cache fetches with the same length L; again, we randomized the time between successive fetches according to an exponential distribution.

We performed two sets of simulations. In Figure 4.5, we keep a constant value of 30% for the peripheral load and

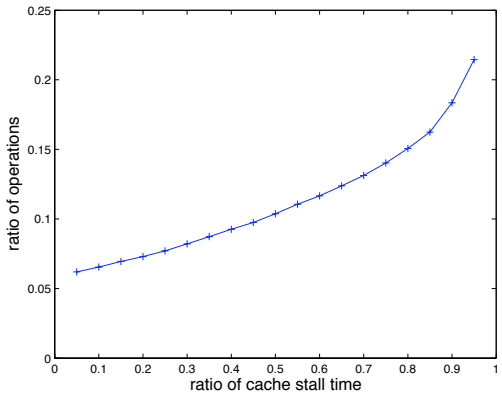


Figure 4.7: Overhead of Alg. 2 with respect to Alg. 1.

we vary the ratio of cache stall time, i.e. the percentage of time that the processor is stalled waiting for a cache line fetch in $c(t)$. In Figure 4.6, we instead keep a constant cache stall time ratio of 20% and we vary the bus load between 2.5% and 50%. In both cases, we performed 1000 simulations for each point. Results are provided as the ratio between the total computed delay $D(t_1, t_N)$ and the WCET of the task, and between $D(t_1, t_N)$ and the total time NL required by cache fetches, which represents the per-fetch increase in stall time. Note that the average per-fetch delay in Figure 4.5 decreases almost linearly as the cache stall time increases. This is expected, since the constraint imposed by $E(t)$ becomes more stringent as the time between successive fetches decreases. However, the delay increases with respect to the WCET of the task, since fetches account for a bigger part of the WCET as the stall time increases. Again, for ratios that are typical in real tasks (5-20%), the increment is almost linear. Figure 4.6 shows a similar behavior, except that the per-fetch delay saturates around a load of 45%. Intuitively, at that point the peripheral interference is high enough that the delay is dominated by the blocking function $B(t)$; note that delay saturates at 75% according to the ratio $L'/L = 3/4$. This result is interesting, because it shows that even moderate amounts of peripheral traffic can actually cause worst case interference.

In Figure 4.7 we compare the computation times of Algorithms 1, 2. The same scenario as in Figure 4.5 is shown; we plot the average ratio of the number of times $\bar{E}(t)$ is computed in Algorithm 2 compared to Algorithm 1. As it can be seen from the figure, by applying selective pruning Algorithm 2 has significantly less overhead, requiring on average only 6-22% of the operations performed by Algorithm 1.

4.3 Multitasking Analysis

We now describe how the analysis introduced in Section 4.2 can be used to derive schedulability conditions in a multitasking environment. We consider a set of N tasks $\{\tau_1, \ldots, \tau_i, \ldots, \tau_N\}$, each of which is associated with possibly

multiple cache profiles $c_{i,j}^{prof}, 1 \leq j \leq M_i$ as detailed in Section 4.1. Each task τ_i has a fixed period p_i and a relative deadline D_i, with $D_i \leq p_i$. Given peripheral load function $E(t)$, we want to determine if the task set is schedulable according to a specified scheduling algorithm. In the remaining of this section, we first detail the simpler case of a cyclic executive scheduler, and then we show how to analyze the more complex case of a run-time dynamic scheduler.

For the cyclic executive scheduler, we assume that each task is assigned a static computational share in addition to period and deadline. A schedule can then be computed where each instance (job) of each task τ_i is divided in K_i computation chunks, each of which must be completed within a time interval of length T_i. The goal of the analysis is therefore the following: given design parameters K_i, T_i, determine if there exists a static partition of τ_i in K_i computation chunks such that each chunk completes in at most T_i time units when interference from peripheral transactions on the memory bus is considered. Since each task is composed of τ_i is composed of S_i non-preemptive scheduling intervals, the static partition assigns a given set of scheduling intervals to each computation chunk.

Algorithm 3 Schedulability($E(t), \{c_{i,j}^{prof}\}, K_i, T_i$)

1: $\{c_{i,j}^{init}\}$:= $\{c_{i,j}^{prof}\}$
2: $init$:= $curr$:= 0
3: **for** $k = 1 \ldots K_i$ **do**
4: T := 0
5: **while** $T \leq T_i$ **do**
6: $curr$:= $curr + 1$
7: **if** $curr$ = $S_i + 1$ **then**
8: return TRUE
9: **end if**
10: compute $T := \max_{1 \leq j \leq M_i}(D(t_{i,j}^{init}, t_{i,j}^{curr}) + t_{i,j}^{curr} - t_{i,j}^{init})$
11: **end while**
12: **if** $curr$=$init + 1$ **then**
13: return FALSE
14: **end if**
15: $init$:= $curr$:= $curr - 1$
16: $\{c_{i,j}^{prof}\}$:= AddCacheMisses($s_{i,curr+1}, \{c_{i,j}^{init}\}$)
17: **end for**
18: **return** FALSE

Given these assumptions, Algorithm 3 shows our proposed sufficient schedulability analysis. In the algorithm, $t_{i,j}^0 = 0$, and $t_{i,j}^k$ is the time at which the end of scheduling interval $s_{i,k}$ is reached in profile $c_{i,j}^{prof}(t)$. The algorithm iterates over the scheduling intervals deriving the maximum number of intervals that can be executed in an time

window of length T_i, by computing maximal delay $D(t_{i,j}^{init}, t_{i,j}^{curr})$ using Algorithm 2. The algorithm returns FALSE if all K_i chunks are used up or if it is impossible to fit a single scheduling interval in one chunk; if it allocates all S_i scheduling intervals successfully, then it returns TRUE.

It remains to explain the meaning of the AddCacheMisses($s_{i,curr+1}, \{c_{i,j}^{init}\}$) call. The effect of partitioning task τ_i at the end of $s_{i,cur}$ is that other tasks are executed between the $curr$ and $curr + 1$ scheduling interval of task τ_i. Therefore, cache lines of τ_i can be evicted due to the interference of other tasks; cache profiles $c_{i,j}^{prof}$ must thus be modified from the beginning of $s_{i,curr+1}$ onwards to account for added cache misses due to evictions. In the remaining of this section, we assume that the whole cache can be invalidated before $s_{i,curr+1}$, resulting in an unknown cache state; this is the only safe assumption when the memory mapping of the interfering tasks is unknown, which is typically the case when tasks are developed independently and the analysis must be run with incomplete information. Since the cache state is unknown, the cache profile for scheduling intervals $curr+1, \ldots, S_i$ is independent from previous cache misses. It is therefore sufficient for the AddCacheMisses($s_{i,curr+1}, \{c_{i,j}^{init}\}$) function to modify the cache profiles from $s_{i,curr+1}$ onwards, and we need to run it at most once for each profile and for each scheduling interval excluding the first one, that is $M_i(S_i - 1)$ times. Variables $\{c_{i,j}^{init}\}$ are used to store the unmodified profiles; times $t_{i,j}^{init}, t_{i,j}^{curr}$ and the fetch start times used by Algorithm 2 are instead taken from the profile modified by AddCacheMisses. The way AddCacheMisses is computed depends on the methodology used to derive cache profiles, for example in the case of testing we can rerun the experiment adding cache invalidation instructions at the end of scheduling interval $curr$. In the worst case, Algorithm 3 will allocate a single scheduling interval in each chunk, calling Algorithm 2 $M_i(2S_i - 1)$ times. Assuming that AddCacheMisses is precomputed $M_i(S_i - 1)$ times before running the algorithm, the worst case complexity of Algorithm 3 is thus $O(M_i S_i N^2 K)$, where N is the maximum number of cache misses in any scheduling interval after the cache has been invalidated. While this complexity can be high, especially for a task with a large number of cache misses, we would like to point out that the algorithm is only intended for off-line analysis. As a final note, if the set of interfering tasks and their memory mapping is known, it is sometimes possible to restrict the set of evicted cache lines of τ_i, resulting in a smaller number of added misses (as an example, see the cache partitioning technique described in [17]).

Cyclic executive schedules are relatively simple to analyze because preemption points can be statically computed. When using an on-line preemptive scheduler such as Rate Monotonic (RM) or Earliest Deadline First (EDF) [58], however, this is no longer true: a task can be preempted at any time. While in theory we could account for it introducing a preemption point after every executed instruction in a task trace (at the cost of much increased complexity in the analysis), an unrestricted preemption model is undesirable from our point of view: if a task is preempted inside a memory-intensive loop, a large number of cache lines could be evicted due to interference of preempting tasks. We therefore consider a restricted-preemption model [13], under which each task τ_i is comprised of non-preemptive chunks of size at most q_i, and preemption can only happen between chunks. In particular, we identify each non-preemptive chunk with a single scheduling interval. Task set schedulability under EDF can then be checked using the

following theorem.

Theorem 13 ([13]) *A restricted-preemption task system is schedulable under EDF if and only if:*

$$\forall t, t \geq 0 : \sum_{i=1}^{N} \mathrm{DBF}(\tau_i, t) \leq t \tag{4.6}$$

and

$$\forall \tau_j, \forall t, t \geq 0 : q_j + \sum_{i=1, i \neq j}^{N} \mathrm{DBF}(\tau_i, t) \leq t, \tag{4.7}$$

where for task τ_i, q_i is the maximum size of any non-preemptive chunks, e_i is the task's WCET, and $\mathrm{DBF}(\tau_i, t)$ is the demand bound function*, that is:*

$$\mathrm{DBF}(\tau_i, t) = \max\Big(0, \Big(\Big\lfloor \frac{t - D_i}{p_i} \Big\rfloor + 1\Big) e_i\Big). \tag{4.8}$$

Note that while the theorem is stated for each $t \geq 0$, in practice it is sufficient to check all points of discontinuity for the demand bound function between 0 and one hyperperiod (the minimum common multiple of the task periods), whose number has been shown to be pseudo-polynomial [14]. Furthermore, note that the same technique we discuss for EDF can be applied to RM or any other on-line scheduler, as long as a schedulability test for the restricted-preemption model can be derived.

It remains to compute q_i and e_i for each task τ_i. Let q_i^k be the worst case execution time, including delay due to peripheral interference, of scheduling interval k of τ_i. Then $q_i = \max_{1 \leq k \leq S_i} q_i^k$. The worst case execution time for scheduling interval k clearly happens when the task is preempted immediately before the scheduling interval starts. Therefore, following the discussion of Algorithm 3, q_i^k can be computed by calling AddCacheMisses($s_{i,k}, \{c_{i,j}^{prof}\}$), after which $q_i^k = \max_{1 \leq j \leq M_i}(D(t_{i,j}^{k-1}, t_{i,j}^k) + t_{i,j}^k - t_{i,j}^{k-1})$.

Computing e_i requires more work. Let Pr_i be the maximum number of preemptions suffered by any job of τ_i, and assume a non-trivial case in which $Pr_i < S_i - 1$. For an on-line scheduler that assigns a fixed priority to each job, like EDF and RM, if tasks can not self-suspend or be blocked, then the number of preemptions suffered by a job of τ_i is at most equal to the number of higher priority jobs that can start while it is active [59]. In particular, for EDF with $D_i \leq p_i$ it holds:

$$Pr_i \leq \sum_{j=1, j \neq i}^{N} \Big\lfloor \frac{D_i + (p_j - D_j)}{p_j} \Big\rfloor \tag{4.9}$$

Since each preemption increases the execution time of τ_i by causing additional cache misses, a safe bound on e_i can be obtained by considering a worst case pattern of Pr_i preemptions over the $S_i - 1$ ending points of scheduling intervals $s_{i,1}, \ldots, s_{i,S_i-1}$. A naive approach would be to run the delay analysis on each of the $\begin{pmatrix} S_i - 1 \\ Pr_i \end{pmatrix}$ preemption patterns, but this is usually unfeasible as the number of patterns is exponential in S_i. Instead, we propose a dynamic programming algorithm that is able to obtain the worst case pattern in time polynomial in S_i, Pr_i.

The algorithm works by iteratively computing values $e_{i,j}^{1,0}, \ldots, e_{i,j}^{S_i,0}, e_{i,j}^{2,1}, \ldots, e_{i,j}^{S_i,1}, \ldots e_{i,j}^{Pr_i+1,Pr_i}, \ldots, e_{i,j}^{S_i,Pr_i}$ for cache pattern $c_{i,j}(t)$: $e_{i,j}^{k,l}$ represents the WCET for the first k scheduling intervals, including delay due to peripheral interference, assuming they suffer l preemptions (note it must hold $0 \leq l < k$). The WCET e_i for the whole task can then be simply obtained as $e_i = \max_{1 \leq j \leq M_i} e_{i,j}^{S_i,Pr_i}$.

Values $e_{i,j}^{k,0}$ can be computed as $e_{i,j}^{k,0} = D(t_{i,j}^0, t_{i,j}^k) + t_{i,j}^k - t_{i,j}^0$ since they have no preemption. All other values are computed according to Algorithm 4.

Algorithm 4 Compute $e_{i,j}^{S_i,Pr_i}(E(t), c_{i,j}^{prof}, \{e_{i,j}^{k,0}\})$

```
1:  c^init := c_{i,j}^prof
2:  for l := 1 to Pr_i do
3:      for k := l + 1 to S_i do
4:          e_{i,j}^{k,l} := 0
5:          for c := l to k − 1 do
6:              c_{i,j}(t) := AddCacheMisses(s_{i,c+1}, c^init)
7:              temp := e_{i,j}^{c,l−1} + D(t_{i,j}^c, t_{i,j}^k) + t_{i,j}^k − t_{i,j}^c
8:              if temp > e_{i,j}^{k,l} then
9:                  e_{i,j}^{k,l} := temp
10:             end if
11:         end for
12:     end for
13: end for
14: return e_{i,j}^{S_i,Pr_i}
```

The algorithm iterates over l, k as explained before. Each value $e_{i,j}^{k,l}$ is obtained based on the previously computed $e_{i,j}^{l,l-1}, \ldots, e_{i,j}^{c,l-1}, \ldots, e_{i,j}^{S_i,l-1}$ values as follows: at each iteration point in the innermost loop, a temporary value is computed summing the $e_{i,j}^{c,l-1}$ value with the worst case delay for the remaining scheduling intervals $c+1, \ldots, k$, assuming that the l-th preemption happens at the end of scheduling interval $s_{i,c}$. $e_{i,j}^{k,l}$ is then computed as the maximum of all such temporary values. Assuming again that AddCacheMisses is precomputed $S_i - 1$ times for profile $c_{i,j}^{prof}$, Algorithm 4 has therefore complexity $O(Pr_i S_i^2 N^2 K)$.

Theorem 14 *Given cache access function $c_{i,j}(t)$ and peripheral load function $E(t)$, $e_{i,j}^{k,l}$ is a safe upper bound on the worst case computation time for scheduling intervals $1, \ldots, k$ of τ_i, assuming that such scheduling intervals are preempted l times.*

Proof.

The proof proceeds by induction on l, k, using the iterative order of Algorithm 4, that is, $e_{i,j}^{c,z} \prec e_{i,j}^{k,l}$ iff $z < l \vee (z =$

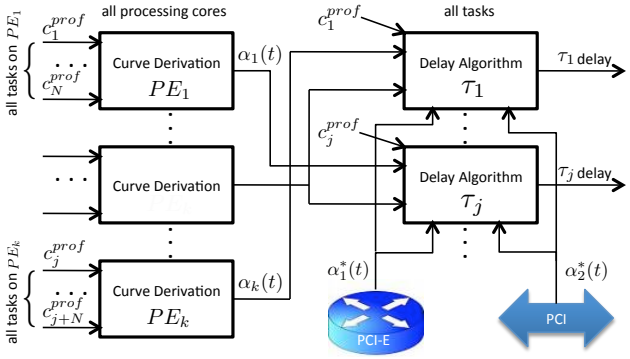


Figure 4.8: Analysis Methodology.

$l \wedge c < k$).

Base Case: We need to show that $e_{i,j}^{k,0} = D(t_{i,j}^0, t_{i,j}^k) + t_{i,j}^k - t_{i,j}^0$ is an upper bound to the WCET for scheduling intervals $1, \ldots, k$ assuming zero preemptions; this follows immediately since $t_{i,j}^k - t_{i,j}^0$ is the execution time of scheduling intervals $1, \ldots, k$ in $c_{i,j}(t)$ and furthermore according to Theorem 12, $D(t_{i,j}^0, t_{i,j}^k)$ is a tight upper bound for the delay suffered by cache misses in those scheduling intervals.

Induction Step: Assuming that the hypothesis holds for values $e_{i,j}^{1,0}, \ldots, e_{i,j}^{k-1,l}$, we need to show that it holds for $e_{i,j}^{k,l}$ as well. Assume that the last point at which scheduling intervals $1, \ldots, k$ suffer a preemption in any pattern of l preemptions is the end of interval $s_{i,c}$. First note that it must hold that $c \leq k-1$, since the task must be preempted before the end of the k-th scheduling interval, and $c \geq l$, since there are $l-1$ preemptions at points $s_i^1, \ldots, s_i^{c-1}$. We will show that the innermost loop of Algorithm 4 computes at each step an upper bound on $e_{i,j}^{k,l}$, assuming that the last preemption is suffered at the end of $s_{i,c}$. This will prove the induction step since the algorithm set $e_{i,j}^{k,l}$ to be the maximum of all such bounds for $l \leq c \leq k-1$.

Since the last preemption happens at the end of $s_{i,c}$, a bound on WCET of scheduling intervals $c+1, \ldots, k$ can be obtained calling AddCacheMisses($s_{i,c+1}, c^{init}(t)$) to account for the additional cache misses due to preemption at the end of $s_{i,c}$, and then computing the execution time, inclusive of peripheral-induced delay, which is equal to $D(t_{i,j}^c, t_{i,j}^k) + t_{i,j}^k - t_{i,j}^c$. Finally, due to the induction hypothesis, $e_{i,j}^{c,l-1}$ is a valid upper bound to the WCET of scheduling intervals $1, \ldots, c$ given that they suffer $l-1$ preemptions. Since the innermost loop of the algorithm simply sums the two bounds together, this concludes the proof. □

4.4 Multicore Analysis

We now extend our analysis to compute delay bounds for tasks running in a multicore system, assuming the interval-based model for cache profiles. We assume that peripheral traffic is buffered in its respective interconnection, and

that multiple interconnections can access main memory: hence, the task under analysis can suffer interference from multiple buffered flows with arrival curves $\alpha_1^*(t), \ldots, \alpha_i^*(t), \ldots$. Furthermore, tasks running on other cores can also interfere with the task under analysis. Since processing cores are not strictly synchronized, while a task τ_i is running on core PE_j we do not know which tasks are running on the other cores. To solve the problem, our analysis follow two successive steps; in particular, we compute a delay bound for each task running the analysis once for every task under analysis on every processing core.

1. For each processing core PE_i, we derive an upper arrival curve to requests made by all tasks running on PE_i. In this way, each processing core is substituted by an *unbuffered flow* with arrival curve $\alpha_i(t)$: in any interval of time t, $\alpha_i(t)$ represents the maximum amount of time required by tasks running on PE_i to perform operations in main memory.

2. The analysis of Section 4.4.2 computes an upper delay bound for the task under analysis given a set F of interfering flows, e.g. all peripheral buffered flows and the unbuffered flows for all processing cores except the one where the task under analysis is running.

A clarifying example is shown in Figure 4.8 for a system with two peripheral interconnections with buffered arrival curves $\alpha_1^*(t), \alpha_2^*(t)$. Finally, we make more general assumptions on the arbitration for access to main memory, assuming that each request for memory access can be split into multiple atomic operations in main memory, which is usually the case in modern memory controllers. Each processing core PE_i is characterized by a parameter C_i, which is the length in time (assumed to be fixed) needed to service a memory request. Furthermore, for each buffered flow and processing core / unbuffered flow we define an arbitration parameter L_i, which is the maximum length of an atomic operation in main memory. Typically, C_i is a integer multiple of L_i. For example, if the size of a cache line and associated fetch request is 128 bytes and memory arbitration is based on 32 bytes operations, then every memory request consists of 4 atomic operations.

The rest of this section is organized as follows. In Section 4.4.1 we show how to derive arrival curves for each unbuffered flow based on the cache profiles of the tasks running on the corresponding processing core. In Section 4.4.2 we detail our analysis and prove its correctness. Finally, in Section 4.4.3 we show how to experimentally derive interval-based cache profiles for tasks, and in Section 4.4.4 we detail experiments and simulation results based on the described analysis. Notation used throughout the section is summarized in Figure 4.9.

4.4.1 Computing Arrival Curves

A task τ_i causes interference to other tasks in the system by accessing main memory according to the pattern defined by c_i^{prof}. We specify this interference as an arrival curve [98] by considering the scheduling intervals of task τ_i and their respective bounds on execution time and accesses to the shared resource. Tasks execute periodically on a processing element PE_i and therefore we can derive an arrival curve that represents a tasks behavior for any time window. We

Symbol	Description
PE_i	i-th processing core
C_i	maximum length of memory request in seconds
L_i	maximum length of atomic operation in seconds
τ_i	i-th task
p_i	period of task τ_i
c_i^{prof}	interval-based cache profile for task τ_i
S_i	number of scheduling intervals of task τ_i
$s_{i,j}$	j-th scheduling interval of task τ_i
$exec_{i,j}^U, exec_{i,j}^L$	upper and lower computation time of $s_{i,j}$ (assuming memory operations take zero time)
$\mu_{i,j}^{\max}, \mu_{i,j}^{\min}$	maximum and minimum number of memory requests
$\tilde{\mu}_{i,j}^{\max}, \tilde{\mu}_{i,j}^{\min}$	maximum and minimum number of replacements
$\alpha_i(t)$	unbuffered arrival curve for i-th flow
$\alpha_i^*(t)$	buffered arrival curve for i-th flow
$\bar{\alpha}_i(t)$	traffic delay curve for i-th flow
b_i	max backlog for i-th buffered flow
$*\tau_i'$	sequence set for two instances of τ_i
$t'_{m,d}$	scheduling interval set $\{s_{i,m}, \ldots s_{i,m+d}\}$
$\gamma^{\max}, \gamma^{\min}$	maximum and minimum number of accesses in time window
$\Delta^{\max^L}, \Delta^{\min^L}$	length of time window with maximum, minimum number of accesses and lower execution time
$D_{j,k}$	maximum delay for scheduling intervals $\{s_j, \ldots, s_k\}$
$D_{j,k}^i$	maximum delay caused by i-th flow
$Ub_{j,k}$	computed upper bound on $D_{j,k}$
$Ub_{j,k}^i$	computed upper bound on $D_{j,k}^i$
$Ub_{j,k}^{(i)}$	upper bound on delay for $\{s_j, \ldots, s_k\}$ caused by all flows **except** i-th flow
$u_k^{i,j}$	delay term for s_k, i-th flow computed based on $\{s_j, \ldots, s_k\}$

Figure 4.9: Section Notation.

initially consider each task in isolation, assuming that no other system component accesses main memory; we then show how to derive an arrival curve for multiple tasks executed on the same core, assuming that tasks are scheduled according to fixed timeslots (e.g. cyclic executive). The derived arrival curves will be later used in Section 4.4.2 to compute a delay bound for the task under analysis. Note that when computing the arrival curve for a core, we do not consider the interference caused by other flows on it. As shown in Section 4.4.2, this is because each atomic operation of the task under analysis can be delayed for its worst case amount while the interfering cores themselves suffer no delay.

Deriving arrival curves involves **(1)** computing all possible sequences of subsequent scheduling intervals, **(2)** computing the feasible time windows for each sequence, **(3)** computing the minimal and maximal number of cache misses for each time window and **(4)** constructing the arrival curves accordingly.

Single Task per Processing Element

In this section we introduce our approach to represent a tasks accesses to a shared resource as arrival curve, assuming a single periodic task per processing element.

Computing sequences of scheduling intervals. Based on the parameters c_i^{prof} of task τ_i, we can derive time windows for which the minimal and maximal number of accesses to a shared resource are known. The time windows can be computed from the set of all possible sequences of subsequent scheduling intervals, i.e., the *sequence set*. As an example, let $\tau_1 = \{s_{1,1}, s_{1,2}\}$ then the sequence set is $*\tau_1 = \{\{s_{1,1}\}, \{s_{1,2}\}, \{s_{1,1}, s_{1,2}\}\}$.

In order to account for the transition phase between succeeding periods of a task, we consider two subsequent instances of a task for the arrival curve derivation. Therefore we specify $\tau_i' = \{\tau_i\ \tau_i\}$, such that $\tau_i' = \{s_{i,1} \dots s_{i,S_i}, s_{i,1} \dots s_{i,S_i}\}$ and $*\tau_i'$ is the corresponding *sequence set*. Element $t'_{m,d} \in *\tau'$ is described by the offset m, representing the index of its first scheduling interval from τ' and d, representing the number of scheduling intervals considered, such that:

$$t'_{m,d} = \{s_{i,m}, \dots, s_{i,m+d}\}\ \forall d \in [0 \dots S_i - 1], \forall m \in [1 \dots S_i]. \tag{4.10}$$

Computing time windows. Each element $t'_{m,d} \in *\tau_i'$ results in four different time windows, considering all combinations of minimal and maximal execution times and accesses to the shared resource respectively. Two of these time windows represent the worst case, i.e., they represent the maximal number of accesses for the shortest time windows. The time windows Δ and their corresponding number of accesses to the shared resource γ are represented as tuples $\hat{t} =< \gamma, \Delta >$ and we show how to compute them in the next subsection. Accesses to the shared resource can happen at any time during a scheduling intervals execution. In other words, for the first and last scheduling interval in a sequence $t'_{m,d}$, the accesses to the shared resource happen at the end and at the beginning respectively. As a result, the first and last scheduling intervals' execution times $exec_{i,j}$ are not considered for the representative time windows but their accesses to the shared resource are considered.

Consider Fig. 4.10 for an example how to compute time windows. In the first example, denoted `1 scheduling`

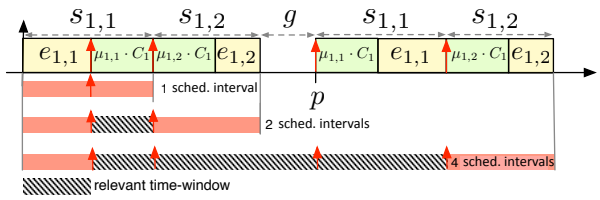


Figure 4.10: Computing time windows for sequences of 1, 2 and 4 scheduling intervals including the gap between periods.

`interval`, the time window computes as zero, meaning that accesses to the shared resource occur concurrently at one instant of time. For example `2 scheduling intervals`, the time window Δ computes as the time required to process the first scheduling intervals accesses, while the number of accesses γ computes as the sum of both scheduling intervals' accesses. Execution times are not considered for the time window, since in the worst case the actual computation is performed before and after the first and second scheduling interval respectively. The first scheduling interval accesses to the shared resource need to be processed before the second scheduling interval can be activated, specifying the time window. In other words, we arrange the accesses to the shared resource in subsequent scheduling intervals such that the resulting time window is minimized, conclusively maximizing the interference onto other tasks.

Computing the time window for elements $t'_{m,d}$, whose scheduling interval sequence spans over the period, needs to consider the gap g between the last scheduling interval of a task and its period. Example `4 scheduling intervals` in Fig. 4.10 illustrates such a case. Minimizing the gap, and deductively the time window, is done by assuming the maximal execution time $exec^U_{i,j}$ and number of accesses $\mu^{max}_{i,j}$ for scheduling intervals not included in $t'_{m,d}$.

$$g(e,r) = p_i - \sum_{\forall s_{i,j} \in \tau_i \setminus (\tau_i \cap t'_{m,d})} exec^U_{i,j} + \mu^{max}_{i,j} \cdot C_i - \sum_{\forall s_{i,j} \in \tau_i \cap t'_{m,d}} exec^e_{i,j} + \mu^r_{i,j} \cdot C_i, \tag{4.11}$$

where e and r denote the actual values for execution time and accesses to the shared resource, e.g., $e = U$ and $r = min$.

Computing the cache misses for each time window. Time windows Δ and the corresponding number of accesses to the shared resource γ for an element $t'_{m,d}$ are computed in Equations 4.12 and 4.13. Based on these values, the tuples for element $t'_{m,d} \in *\tau'$ are computed in Equations 4.14 to 4.15.

$$\gamma^{min} = \sum_{j=m}^{m+d} \mu^{min}_{i,j} \tag{4.12}$$

$$\Delta^{min^L} = \sum_{j=m+1}^{m+d-1} exec^L_{i,j} + \sum_{j=m}^{m+d-1} \mu^{min}_{i,j} \cdot C_i \tag{4.13}$$

$$\hat{t}^{min^L}_{m,d} =< \gamma^{min} \; ; \; \Delta^{min^L} + g(L, min) > \tag{4.14}$$

$$\hat{t}_{m,d}^{max^L} =< \gamma^{max} \; ; \; \Delta^{max^L} + g(L, max) > \tag{4.15}$$

Equation 4.14 can be transformed into the tuple computed with Equation 4.15 by simply increasing $\mu_{i,j}^{min}$ to $\mu_{i,j}^{max}$. In other words, they show a linear relation, since the time required to process an access is constant and intermediate tuples can be computed by linear approximation. For any number of accesses to the shared resource within the range of the tuples computed in Equations 4.14 and 4.15, we can therefore compute a safe upper bound to the number of accesses performed in the corresponding time window by linear approximation, see Fig. 4.11.

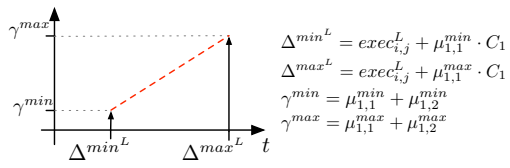


Figure 4.11: Linear approximation between minimum and maximum number of accesses to the shared resource for a single scheduling interval.

Deriving arrival curves. Retrieving the minimal and maximal number of accesses to the shared resource for every time interval $\Delta = \{0 \dots 2p_i\}$ from the computed tuples and linear approximations allows to compute the arrival curve. Consider the function $\delta(\hat{t})$ to return the length of the time window and $\nu(\hat{t})$ to return the number of cache misses for each tuple, then the upper arrival curve $\tilde{\alpha}_i$ can be obtained as:

$$\tilde{\alpha}_i(\Delta) = \underset{\forall \hat{t}_{m,d}; \delta(\hat{t}_{m,d}) = \Delta}{\operatorname{argmax}} \nu(\hat{t}_{m,d}). \tag{4.16}$$

We construct the infinite curves $\hat{\alpha}_i$ as an initial aperiodic part, that is represented by $\widetilde{\alpha}_i$ and a periodic part which is repeated k-times for $k \in \mathbb{N}$.

$$\hat{\alpha}_i(\Delta) = \begin{cases} \tilde{\alpha}_i(\Delta) & 0 \leq \Delta \leq p \\ \max \left\{ \widetilde{\alpha}_i(\Delta), \tilde{\alpha}_i(\Delta - p_i) + \sum\limits_{\forall j} (\mu_{i,j}^{max}) \right\} & p_i \leq \Delta \leq 2p \\ \widetilde{\alpha}_i(\Delta - k \cdot p_i) + k \sum\limits_{\forall j} (\mu_{i,j}^{max}) & \text{otherwise} \end{cases} \tag{4.17}$$

The computational complexity to obtain the overall arrival curves is $O(S_i^2)$. Following the previous computation we derive Lemma 15.

Lemma 15 *Deriving alpha curve $\hat{\alpha}_i(\Delta)$ by Equation 4.17 is the upper bound of accesses to a shared resource by task τ_i for any time window Δ.*

Arrival curve $\alpha_i(t)$ represents the maximum amount of time a task τ_i requires to perform its accesses to the shared resource in a time window of length t and is obtained as $\alpha_i(t) = \hat{\alpha}_i(t) \cdot C_i$.

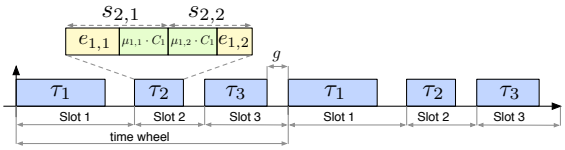


Figure 4.12: Example of 3 periodic tasks executing in a static time wheel

Multiple Tasks per Processing Element

In this section we show how to extend the previously shown approach to multiple tasks executing on each processing element. Consider a set of periodic tasks $T = \{\tau_1 \dots \tau_N\}$ scheduled statically, e.g., a static time slot assignment as in Fig. 4.12. Then we can compute a sequence of all scheduling intervals that constitute tasks in T as:

$$\sigma = \{s_{1,1} \dots s_{1,S_1}, s_{2,1} \dots s_{2,S_2} \dots s_{N,1} \dots s_{N,S_N}\} \tag{4.18}$$

Based on σ the *sequence set* $*\sigma$ is derived, as shown previously. We define $\sigma' = \{\sigma\ \sigma\}$ and compute the sequence set $*\sigma'$ such that $t'_{m,d} \in *\sigma'$, resulting in two periods of the static time wheel being considered.

Time windows are computed from the sequence set $*\sigma'$ following the concept presented for a single task per processing element. We compute the maximal and minimal time windows for any possible sequence of subsequent scheduling intervals in $*\sigma'$ and count the corresponding minimal and maximal number of accesses to the shared resource respectively. Elements $t'_{m,d} \in *\sigma'$ contain scheduling intervals from different statically scheduled tasks and therefore the gap between each tasks' last scheduling interval and its period has to be considered. Similarly to the gap g for the single task approach, minimizing the gap results in the worst case.

Equation 4.11 can be rewritten to compute the gap for a sequence of scheduling intervals, by maximizing all the scheduling intervals that are not considered by element $t'_{m,d}$:

$$g(e,r) = \sum_{\forall \tau_i \in t'_{m,d}} p_i - \sum_{\forall s_{i,j} \in \sigma \setminus (\sigma \cap t'_{m,d})} exec^U_{i,j} + \mu^{max}_{i,j} \cdot C_i - \sum_{\forall s_{i,j} \in \sigma \cap t'_{m,d}} exec^e_{i,j} + \mu^r_{i,j} \cdot C_i. \tag{4.19}$$

The tuples can now be computed as shown in Equations 4.14 to 4.15 and based on them the arrival curve can be derived as shown for the single task case.

4.4.2 Delay Analysis

The goal of the analysis is the derivation of worst case delay that the task under analysis can suffer due to memory contention given a set F of interfering flows. For simplicity, we only show our analysis assuming that the task under analysis runs in isolation on its assigned processing core. The algorithms of Section 4.3 can be applied to compute upper delay bound for multiple tasks running on the same core, substituting the analysis of Section 4.2 with the one in this section.

To simplify notation, we again drop the subscript denoting the number of the task under analysis and use c^{prof} as its cache profile, $\{s_1, \ldots, s_S\}$ as its scheduling intervals and C, L as the parameters for its core. We also use $i \in F$ in place of $\alpha_i \in F$ or $\alpha_i^* \in F$. Let $D_{j,k}$ be the maximum delay suffered by the task in scheduling intervals $\{s_j, \ldots, s_k\}$; the overall task delay is equal to $D_{1,S}$. Similar to Section 4.2, the analysis is based on the following main idea: we first compute an upper bound $Ub_{j,k}$ to the maximum delay $D_{j,k}$ for all $j, k : 1 \leq j \leq k \leq S$, meaning $D_{j,k} \leq Ub_{j,k}$. We then progressively decrease the bound by taking the intersection of multiple such constraints.

Since multiple flows contend with the task under analysis for access to main memory, we divide the delay contribution among all interfering flows. To be more precise, assume that an atomic memory operation by the task under analysis is requested at time t' and first serviced at time t''. Since we only consider work-conserving memory arbitration schemes, it follows that one or more interfering flows must be serviced in interval $[t', t'']$. Suppose that a flow α_i is serviced for Δ time units in $[t', t'']$; then we say that α_i has delayed the task under analysis Δ time units for that memory operation. Based on this definition, we use $D_{j,k}^i$ to denote the maximum total delay caused by flow α_i (or α_i^*) on all operations in scheduling intervals $\{s_j, \ldots, s_k\}$ and $Ub_{j,k}^i$ for its upper bound. We can then obtain $Ub_{j,k}$ as follows:

$$Ub_{j,k} = \sum_{i \in F} Ub_{j,k}^i. \tag{4.20}$$

Delay bound derivation depends on the memory arbitration scheme. In details, we assume that arbitration among the task under analysis and other unbuffered flows follows either a RR or FCFS policy, while arbitration among those and buffered flows follows either RR, FCFS or FP with buffered flows being assigned lowest priority[2]. Arbitration effects are captured by the following lemma.

Lemma 16 *Under RR arbitration (or FP for unbuffered flows with lowest priority), each atomic memory operation of the task under analysis can be delayed by flow α_i (or α_i^*) for at most L_i. Under FCFS arbitration, an unbuffered flow can delay each atomic operation for at most C_i, while a buffered flow can delay it for at most b_i, where b_i is the maximum time required to service the backlog (buffered data) of the flow.*

Proof.

Let t' be the time at which an atomic operation of the task under analysis is requested, and t'' be the time at which it is first serviced. Under RR arbitration, the worst case interference is produced when an atomic operation of α_i (α_i^*) is serviced in $[t', t'']$, resulting in a delay of L_i time units. Note that no more than one atomic operation of α_i (α_i^*) can finish in $[t', t'']$, since the priority of the flow immediately becomes lower than the priority of the task under analysis upon finishing an atomic operation.

Consider FP arbitration, with buffered flow α_i^* having lower priority than the task under analysis. The worst case interference is caused when an atomic operation of α_i^* is first serviced at $t' - \epsilon$, with $\epsilon > 0$, resulting in a delay of $L_i - \epsilon$. Note that no operation of α_i^* can be first serviced in $[t', t'']$, since the flow has lower priority.

[2]This is a common optimization in system controllers for embedded systems, see [63].

Now consider first-come-first-served arbitration for an unbuffered flow α_i. The worst case interference is produced when the task/tasks generating α_i perform a memory request, consisting of C_i/L_i atomic operations, before t' and all C_i/L_i operations are serviced in $[t', t'']$ in C_i time units. Note that since the task/tasks stall while waiting for the last atomic operation to complete, no more than C_i/L_i outstanding operations can be serviced in $[t', t'']$. Finally, for a buffered flow α_i^* the worst case interference is generated when the entire backlog is requested before t' and serviced in $[t', t'']$; since by definition b_i is an upper bound to the time required to service the backlog, the lemma follows. □

Note that in a scheduling interval s_j, the number of atomic memory operations performed by the task under analysis is at most $\mu_j^{\max}\frac{C}{L}$. We can capture the arbitration property expressed by Lemma 16 introducing a *blocking function*[3] B_j^i.

$$B_j^i \equiv \begin{cases} \mu_j^{\max}\frac{C}{L}L_i & \text{for RR and FP} \\ \mu_j^{\max}\frac{C}{L}C_i & \text{for FCFS, unbuffered flow} \\ \mu_j^{\max}\frac{C}{L}b_i & \text{for FCFS, buffered flow} \end{cases} \tag{4.21}$$

We can then express a first upper delay bound in the same way as in Lemma 4 in Section 4.2:

Lemma 17 Blocking Delay Bound: *For each flow and scheduling intervals* $\{s_j, \ldots, s_k\}$*:*

$$D_{j,k}^i \leq \sum_{p=j}^{k} B_p^i \tag{4.22}$$

Proof.

Consider RR arbitration. According to Lemma 16, the maximum delay caused by flow α_i (α_i^*) on an atomic operation of the task under analysis is L_i. The maximum number of atomic operations in scheduling intervals $\{s_j, \ldots, s_k\}$ is $\sum_{p=j}^{k} \mu_p^{\max}\frac{C}{L}$. Therefore, the maximum delay $D_{j,k}^i$ caused by α_i (α_i^*) in scheduling intervals $\{s_j, \ldots, s_k\}$ can not be greater than $(\sum_{p=j}^{k} \mu_p^{\max}\frac{C}{L})L_i = \sum_{p=j}^{k} B_p^i$.

The proof for the other arbitration cases is similar. □

As for Lemma 4, the bound expressed by Lemma 17 is not tight, because each flow might not present enough traffic to cause maximum delay to the task under analysis. We can refine the bound by expressing a condition on the amount of service time required by each flow in scheduling intervals $\{s_j, \ldots, s_k\}$. For simplicity, let $\Delta_{j,k}^{\max^U} \equiv \sum_{p=j}^{k}(exec_p^U + \mu_p^{\max}C)$, e.g. $\Delta_{j,k}^{\max^U}$ is the maximum time required to executed scheduling intervals $\{s_j, \ldots, s_k\}$ with no flow interference.

Lemma 18 *Consider a flow* α_i*, and let* $Ub_{j,k}$ *be an upper bound to the total delay suffered by the task under analysis in scheduling intervals* $\{s_j, \ldots, s_k\}$*. Then:*

$$D_{j,k}^i \leq \alpha_i\big(\Delta_{j,k}^{\max^U} - C + Ub_{j,k}\big) \tag{4.23}$$

[3]Note that the blocking term is much larger for FCFS arbitration than RR. This shows that the fairness added by FCFS is counterproductive in the determination of worst-case guarantees.

Proof.

Let t_j be the start time of scheduling interval s_j and t_{k+1} be the end time of scheduling interval s_k, modified by contention for access to main memory. Furthermore, let t' be the time the first memory operation is requested and t'' be the time the last memory request is serviced in scheduling intervals $\{s_j, \ldots, s_k\}$. Since $Ub_{j,k}$ is an upper bound to the delay suffered by the task in $\{s_j, \ldots, s_k\}$, it holds: $t_{k+1} - t_j \leq \Delta_{j,k}^{\max^U} + Ub_{j,k}$. Furthermore, note that $t' \geq t_j$ and $t'' \leq t_{k+1} - C$ since the last request takes C time to complete and must be finished before the end of s_k. It follows: $t'' - t' \leq \Delta_{j,k}^{\max^U} - C + Ub_{j,k}$. By definition, $D_{j,k}^i$ can not be greater than the total amount of traffic generated by α_i in interval $[t', t'']$, hence the lemma follows. □

Lemma 18 holds for unbuffered flows because $\alpha_i(t'' - t')$ represents the maximum amount of service received in interval $[t', t'']$. However, the same assumption is not true for a buffered flow, since at time t' there can be additional buffered data (backlog), hence the total amount of service time required in $[t', t'']$ can be greater than $\alpha_i(t'' - t')$. The problem can be solved by replacing each buffered flow arrival curve $\alpha_i^*(t)$ with a new arrival curve $\alpha_i(t) = \alpha_i^*(t) + b_i$, where b_i is the maximum time required to service the backlog. Lemma 18, as well as the remaining theorems in this section, can then be applied to both unbuffered and buffered flows using arrival curve α_i. We detail the computation of b_i at the end of this section.

There is a remaining issue. Based on Equation 4.20, the $Ub_{j,k}$ term in Lemma 18 depends on the delay bound $Ub_{j,k}^i$ for flow α_i, but in turn we would like to compute $Ub_{j,k}^i$ based on the delay bound for $D_{j,k}^i$ provided by Lemma 18. To solve this mutual dependency problem, we introduce a new *traffic delay curve* $\bar{\alpha}_i(t)$:

$$\bar{\alpha}_i(t) \equiv \max\{\Delta | \Delta = \alpha_i(t + \Delta)\}. \tag{4.24}$$

$\bar{\alpha}_i(t)$ has the same intuitive meaning as $\bar{E}(t)$ in Section 4.2. We can then obtain $Ub_{j,k}^i$ according to the following lemma, which is the equivalent of Lemma 5 in Section 4.2.

Lemma 19 Traffic Delay Bound: *Consider flow α_i, and let $Ub_{j,k}^{(i)} \equiv \sum_{p \in F, p \neq i} Ub_{j,k}^p$ be an upper bound to the total delay in scheduling intervals $\{s_j, \ldots, s_k\}$ caused to the task under analysis by all flows except flow α_i. Then:*

$$Ub_{j,k}^i = \bar{\alpha}_i\big(\Delta_{j,k}^{\max^U} - C + Ub_{j,k}^{(i)}\big) \tag{4.25}$$

is a valid upper bound to $D_{j,k}^i$.

Proof.

Let $Ub_{j,k}^i$ be an upper bound to $D_{j,k}^i$. Then since $Ub_{j,k}^{(i)} + Ub_{j,k}^i$ is a valid upper bound to $D_{j,k}$, from Lemma 18 it immediately follows that $\sup\{\Delta | \Delta \leq \alpha_i(\Delta_{j,k}^{\max^U} - C + Ub_{j,k} + \Delta)\}$ is a valid value for $Ub_{j,k}^i$.

To prove the lemma it is then sufficient to show that $\forall t, \sup\{\Delta | \Delta = \alpha_i(t + \Delta)\} = \max\{\Delta | \Delta = \alpha_i(t + \Delta)\}$, which is trivially true if the maximum exists for all t. Note that this is the case because of the following properties

for any valid arrival curve $\alpha_i(t)$: 1) $\lim_{t\to\infty} \alpha_i(t)/t < 1$ (otherwise the generating task/tasks would not execute any instruction, or the generating peripheral would continuously occupy the bus); 2) $\alpha_i(t)$ is non-decreasing; 3) $\alpha_i(t)$ is right-continous and has a finite number of discontinuities in any finite time interval. □

A better bound $Ub_{j,k}$ can be obtained by combining Lemmas 17, 19. In particular, the following theorem trivially holds.

Theorem 20 *$Ub^i_{j,k}$ is a valid upper bound to $D^i_{j,k}$, where:*

$$Ub^i_{j,k} = \min\Big(\sum_{p=j}^{k} B^i_p, \bar{\alpha}_i\big(\Delta^{\max^U}_{j,k} - C + Ub^{(i)}_{j,k}\big)\Big) \tag{4.26}$$

Once again, as in Section 4.2, the $Ub_{j,k}$ bound computed according Theorem 20 is a valid upper bound but it is fairly pessimistic. We can therefore refine the bound on $D_{j,k}$ by considering the minimum between $Ub_{j,k}$ and $Ub_{j,q} + Ub_{q+1,k}$. As an additional example, consider a task with three scheduling intervals, $exec^U_1 = 11, exec^U_2 = 29, exec^U_3 = 11, L = C = 1$, and a single flow with $\bar{\alpha}_1(t) = \frac{2}{5}t, B^1_1 = 9, B^1_2 = 2, B^1_3 = 9$; scheduling intervals 1 and 3 are short and have many cache misses (example: cold misses due to calling new functions) while scheduling interval 2 is longer and experiences few cache misses (example: a cycle). Then by computing $Ub_{1,1} + Ub_{2,2} + Ub_{3,3}$ we obtain a delay bound of $\bar{\alpha}_1(10) + B^1_2 + \bar{\alpha}_1(10) = 10$, while computing $Ub_{1,3}$ yields a bound of $B^1_1 + B^1_2 + B^1_3 = \bar{\alpha}_1(11 + 29 + 11 - 1) = 20$.

The example shows that to obtain a better bound on $D_{j,k}$, multiple subintervals might need to be analyzed, but the number of possible sets of subintervals is exponential in $k-j$. Luckily, as in Section 4.2, there is no need to analyze an exponential number of subinterval: rather, it is possible to use an algorithm that only analyzes a quadratic number of subintervals. We apply the same main idea: we compute a delay term $u^{i,j}_k$ for each scheduling interval s_k by iteratively checking all scheduling intervals in $\{s_j, \ldots, s_k\}$. The algorithm iterates over j, k, obtaining at each iteration a bound $Ub^i_{j,k} = \sum_{p=j}^{k} u^{i,j}_p$, which uses the newly computed $u^{i,j}_k$ term. More in details, $u^{i,j}_k$ is computed based on the blocking term B^i_k, the delay terms $u^{i,j}_j, \ldots, u^{i,j}_{k-1}$ and the traffic delay bound of Lemma 19 for each subinterval $\{s_q, \ldots, s_k\}$ with $j \le q \le k$. By computing a delay term $u^{i,j}_k$ for each interfering flow i, the described main idea allows us to produce a tighter bound than Lemmas 17, 19. However, handling multiple flows has an added complexity: when at iteration j, k we compute the traffic delay bound for flow α_i in any subinterval $\{s_q, \ldots, s_k\}$ according to Lemma 19, we must know the maximum delay $Ub^{(i)}_{q,k}$ caused by other flows in that subinterval. The problem can be solved using a dynamic programming approach: instead of iterating over j, k, we first compute delay bounds $Ub_{j,j} = \sum_{i\in F} Ub^i_{j,j}$ for all $j : 1 \le j \le S$, then we compute a delay bound $Ub_{j,j+1}$ for all $j : 1 \le j < S$, then $Ub_{j,j+2}$ and so on and so forth. As shown in Algorithm 5, this is done by iterating over variables d, j, obtaining at each step delay bounds $Ub^i_{j,j+d}$, with $k = j + d$.

Delay term $u^{i,j}_k$ is computed in Equation 4.27 as the minimum of three delay terms: **(1)** term B^i_k for the blocking delay bound of Lemma 17; **(2)** the minimum over all subinterval $\{s_q, \ldots, s_k\}$, with $j + 1 \le q \le k$, for the traffic

Algorithm 5 Compute $\{Ub_{j,k}\}$

1: $\forall i \in F, \forall j, 1 \leq j \leq S : Ub^i_{j,j-1} := 0$

2: $\forall i \in F, \forall j, 1 \leq j \leq S : Ub^{(i)}_{j,j-1} := 0$

3: **for** $d = 0 \ldots S-1$ **do**

4: **for** $j = 1 \ldots S-c$ **do**

5: $k := j + d$

6: solve the following system of equations $\forall i \in F$:

$$u^{i,j}_k = \min\Big(B^i_k, \min_{q:j+1\leq q\leq k}\big\{\bar{\alpha}_i\big(\Delta^{\max^U}_{q,k} - C + Ub^{(i)}_{q,k}\big) - \sum_{p=q}^{k-1} u^{i,j}_p\big\}, \tag{4.27}$$

$$\bar{\alpha}_i\big(\Delta^{\max^U}_{j,k} - C + Ub^{(i)}_{j,k-1} + \sum_{p\in F, p\neq i} u^{p,j}_k\big) - \sum_{p=j}^{k-1} u^{i,j}_p\Big)$$

7: $u^j_k := \sum_{i\in F} u^{i,j}_k$

8: $\forall i \in F : Ub^i_{j,k} := Ub^i_{j,k-1} + u^{i,j}_k$

9: $\forall i \in F : Ub^{(i)}_{j,k} := Ub^{(i)}_{j,k-1} + u^j_k - u^{i,j}_k$

10: $Ub_{j,k} := \sum_{i\in F} Ub^i_{j,k}$

11: **end for**

12: **end for**

13: **return** $\{Ub_{j,k}\}$

delay bound of Lemma 19 (the case of $q = j$ is covered in the third term). Assume that this term becomes minimum among all three terms for a specific choice of q. Then Equation 4.27 can be rewritten as:

$$\sum_{p=q}^{k-1} u^{i,j}_p + u^{i,j}_k = \sum_{p=q}^{k} u^{i,j}_p = \bar{\alpha}_i(\Delta^{\max^U}_{q,k} - C + Ub^{(i)}_{q,k}), \tag{4.28}$$

where following Lemma 19, the rightmost part is an upper bound to $D^i_{q,k}$. Note that since $q \geq j+1$, it holds $k-q < d$, thus the $Ub^{(i)}_{q,k}$ values have already been computed by the algorithm. Also note that for $d = 0$, $j = k$ and there is no valid value for q, so the term is ignored altogether. **(3)** The traffic delay bound for scheduling intervals $\{s_j, \ldots, s_k\}$. This is a special case of the second term for $q = j$: in the equation, $Ub^{(i)}_{j,k-1} + \sum_{p\in F, p\neq i} u^{p,j}_k = Ub^{(i)}_{j,k}$, but we can not directly use $Ub^{(i)}_{j,k}$ because the $u^{p,j}_k$ values need to be computed together with $u^{i,j}_k$. Hence, we actually need to solve a system of equations computing values $u^{i,j}_k$ for all $i \in F$ simultaneously.

Finally, Lines 7-10 are used to compute all delay bounds in $O(\sharp F)$ at each step, where $\sharp F$ is the number of interfering flows. Note that the definitions in Lines 1-2 are required to make sure that the value of $Ub^i_{j,k-1}, Ub^{(i)}_{j,k-1}$ in Equation 4.27 and Lines 8-9 are zero when $d = 0$. Since updating all delay variables takes linear time in the number of interfering flows, the algorithm complexity is dominated by the complexity of solving the system of Equation 4.27, which must be done $O(S^2)$ times. A discussion of how the system can be solved and its complexity is provided in

Section 4.4.2. We can now show that Algorithm 5 computes a valid upper bound $Ub_{j,k}$ to the delay $D_{j,k}$ in any scheduling interval interval $\{s_j, \ldots, s_k\}$.

Theorem 21 *Assume that the system of Equations 4.27 always admits solution. Then for each scheduling interval interval $\{s_j, \ldots, s_k\}$, Algorithm 5 computes a valid upper bound $Ub_{j,k}$ to $D_{j,k}$.*

Proof.

We prove the theorem by induction on d. In particular, we show that $\forall i, \forall j, 1 \leq j \leq S-d : Ub^i_{j,j+d}$ is a valid upper bound to $D^i_{j,j+d}$, from which it follows that $Ub_{j,j+d} = \sum_{p \in F} Ub^p_{j,j+d}$ is an upper bound to $D_{j,j+d}$.

Induction Step: Assume that $\forall q, 0 \leq q < d, \forall i, \forall j, 1 \leq j \leq S-q : Ub^i_{j,j+q}$ is a valid upper bound to $D^i_{j,j+q}$. We have to prove that $\forall i, j : Ub^i_{j,k}$, with $k = j+d$, is an upper bound to $D^i_{j,k}$. By contradiction, assume that $\exists i, j : Ub^i_{j,k}$ is not an upper bound to $D^i_{j,k}$. Then since according to Line 8 $Ub^i_{j,k} = Ub^i_{j,k-1} + u^{i,j}_k$, it follows that at least one of the following two assertions is true for any pattern of memory operations and flow traffic that produces a delay greater than $Ub^i_{j,k}$: the delay suffered by scheduling intervals $\{s_j, \ldots, s_{k-1}\}$ due to interference caused by flow α_i is strictly greater than $Ub^i_{j,k-1}$; or the delay suffered by s_k due to interference of α_i is strictly greater than $u^{i,j}_k$. However, since $k-1-j < d$, the first assertion is impossible due to the induction hypothesis. Hence, let the delay suffered by s_k due to interference of α_i be $u^{i,j}_k + \Delta$, with $\Delta > 0$. We consider three cases, based on which of the three terms in Equation 4.27 is minimum for $u^{i,j}_k$.

1. $u^{i,j}_k = B^i_k$: then applying Lemma 17 to $D^i_{k,k}$ it follows $\Delta = 0$, a contradiction.

2. Let the second term of Equation 4.27 be minimal for q, e.g. $u^{i,j}_k = \bar{\alpha}_i\big(\Delta^{\max^U}_{q,k} - C + Ub^{(i)}_{q,k}\big) - \sum_{p=q}^{k-1} u^{i,j}_p$. Note that the computation of $Ub^{(i)}_{j,k}$ in Lines 7, 9 of the algorithm is equivalent to computing $Ub^{(i)}_{j,k} = \sum_{p \in F, p \neq i} Ub^p_{j,k}$. Since $q \geq j+1$, it holds $k-q < d$, hence by the induction hypothesis $Ub^{(i)}_{q,k}$ is a valid upper bound to the delay caused by all flows except α_i on scheduling intervals $\{s_q, \ldots, s_k\}$. Therefore, according to Lemma 19, the delay caused by α_i in scheduling intervals $\{s_q, \ldots, s_k\}$ can not be greater than $\bar{\alpha}_i\big(\Delta^{\max^U}_{q,k} - C + Ub^{(i)}_{q,k}\big)$. Now note that we can rewrite the second term of Equation 4.27 as $u^{i,j}_k + \sum_{p=q}^{k-1} u^{i,j}_p = \bar{\alpha}_i\big(\Delta^{\max^U}_{q,k} - C + Ub^{(i)}_{q,k}\big)$: since we assumed that the delay in s_k is $u^{i,j}_k + \Delta$, it follows that the delay in $\{s_q, \ldots, s_{k-1}\}$ is at most $\sum_{p=q}^{k-1} u^{i,j}_p - \Delta$. Following Line 8 in the algorithm, $Ub^i_{j,k-1} = Ub^i_{j,q-1} + \sum_{p=q}^{k-1} u^{i,j}_p$; due to the induction hypothesis, $Ub^i_{j,q-1}$ is an upper bound to the delay caused by α_i in $\{s_j, \ldots, s_{q-1}\}$, hence the total delay suffered in $\{s_j, \ldots, s_{k-1}\}$ is at most $Ub^i_{j,k-1} - \Delta$. This contradicts the hypothesis that $Ub^i_{j,k-1} + u^{i,j}_k$ is not an upper bound to $D^i_{j,k}$.

3. Finally, assume that $u^{i,j}_k + \sum_{p=j}^{k-1} u^{i,j}_p = \bar{\alpha}_i\big(\Delta^{\max^U}_{j,k} - C + Ub^{(i)}_{j,k-1} + \sum_{p \in F, p \neq i} u^{p,j}_k\big)$. We prove that $\bar{\alpha}_i\big(\Delta^{\max^U}_{j,k} - C + Ub^{(i)}_{j,k-1} + \sum_{p \in F, p \neq i} u^{p,j}_k\big)$ is an upper bound to the delay caused by α_i in scheduling intervals $\{s_j, \ldots, s_k\}$, which contradicts the hypothesis that $Ub^i_{j,k} = u^{i,j}_k + \sum_{p=j}^{k-1} u^{i,j}_p$ is not an upper bound to $D^i_{j,k}$. This is possible applying Lemma 19 if we can show that $Ub^{(i)}_{j,k-1} + \sum_{p \in F, p \neq i} u^{p,j}_k = \sum_{p \in F, p \neq i} (Ub^p_{j,k-1} + u^{p,j}_k)$ is an upper bound to the delay caused by all flows except α_i in $\{s_j, \ldots, s_k\}$.

Consider flow $\alpha_p, p \neq i$. There are two possible cases: a) $u_k^{p,j}$ is computed based on either term (1) or (2) in Equation 4.27; b) $u_k^{p,j}$ is computed based on the third term. For case a), we can apply the same reasoning as in the previous two points to show that $Ub_{j,k}^p$ is an upper bound to the delay $D_{j,k}^p$. For case b), let $\mathbb{F}$ be the set of all flows (including α_i) that are computed according to the third term. We note the following: 1) according to Equation 4.27, each delay term $u_k^{p,j}, p \in \mathbb{F}$ is monotone non-decreasing in each other delay term in $\mathbb{F}$; 2) due to the definition of $\bar{\alpha_i}(t)$, each $u_k^{p,j}$ term is computed as the maximum value for which the system of equations hold; 3) the system admits solution by hypothesis. This implies that $\forall p \in \mathbb{F} : Ub_{j,k}^p = Ub_{j,k-1}^p + u_k^{p,j}$ is an upper bound to the delay caused by α_p in $\{s_j, \ldots, s_k\}$, which concludes the induction step.

Base Case: For $d = 0$, we have to prove that $\forall i, \forall j, 1 \leq j \leq S : Ub_{j,j}^i$ is a valid upper bound to $D_{j,j}^i$. Note that according to the algorithm, the delay term for α_i at iteration j is computed as $u_j^{i,j} = \min\Big(B_j^i, \bar{\alpha}_i\big(\Delta_{j,j}^{\max^U} - C + \sum_{p \in F, p \neq i} u_j^{p,j}\big)\Big)$. The same arguments as in the induction step for terms (1) and (3) in Equation 4.27 can be used to prove that $Ub_{j,j}^i = u_{j,j}^i$ is a valid upper bound.

□

Solving the Delay System

We now detail how to solve the system of Equation 4.27. Due to the third term in Equation 4.27, each $u_k^{i,j}$ value depends on all other $u_k^{p,j}$ values, which must thus be computed at the same step. We can obtain a solution for all $u_k^{i,j}$ terms using a recurrence: the idea is to start from a vector of values $\vec{u}_k^j(0) = (u_k^{1,j}(0), \ldots, u_k^{i,j}(0), \ldots)$, where $\forall i \in F, u_k^{i,j}(0) \geq u_k^{i,j}$. At each step of the recurrence we then compute a new vector $\vec{u}_k^j(c+1) = (u_k^{1,j}(c+1), \ldots, u_k^{i,j}(c+1), \ldots)$ based on the previous vector $\vec{u}_k^j(c) = (u_k^{1,j}(c), \ldots, u_k^{i,j}(c), \ldots)$ and we show that the series converges to a fixed point that is equal to $\vec{u}_k^j = (u_k^{1,j}, \ldots, u_k^{i,j}, \ldots)$, e.g. it is the only solution to Equation 4.27.

The initial element of the series is obtained as follows:

$$u_k^{i,j}(0) = \min\Big(B_k^i, \min_{q:j+1 \leq q \leq k}\big\{\bar{\alpha}_i\big(\Delta_{q,k}^{\max^U} - C + Ub_{q,k}^{(i)}\big) - \sum_{p=q}^{k-1} u_p^{i,j}\big\}\Big); \tag{4.29}$$

intuitively, we compute the minimum of terms (1) and (2) in Equation 4.27. Each successive element of the series is computed based on term (3):

$$u_k^{i,j}(c+1) = \min\Big(u_k^{i,j}(c), \bar{\alpha}_i\big(\Delta_{j,k}^{\max^U} - C + Ub_{j,k-1}^{(i)} + \sum_{p \in F, p \neq i} u_k^{p,j}(c)\big) - \sum_{p=j}^{k-1} u_p^{i,j}\Big). \tag{4.30}$$

Note that the $Ub_{j,k-1}^{(i)}$ and $u_p^{i,j}$ values are not part of the recurrence because they have already been computed in Algorithm 5 at a previous step. Theorem 26 shows that the iteration is correct. In what follows, given any two delay vectors $\vec{u}_k'^j = (u_k'^{1,j}, \ldots, u_k'^{i,j}, \ldots), \vec{u}_k''^j = (u_k''^{1,j}, \ldots, u_k''^{i,j}, \ldots)$, we shall write $\vec{u}_k'^j \geq \vec{u}_k''^j$ iff $\forall i \in F, u_k'^{i,j} \geq u_k''^{i,j}$ (the other comparison operators are similarly defined, in particular, $\vec{u}_k'^j > \vec{u}_k''^j$ iff $\vec{u}_k'^j \geq \vec{u}_k''^j \wedge \vec{u}_k'^j \neq \vec{u}_k''^j$). The following Lemmas 22-25 prove important facts that are needed in the proof of Theorem 26.

Lemma 22 *In the iteration defined by Equations 4.29, 4.30,* $\forall c \geq 0 : \vec{u}_k^j(c) \geq \vec{u}_k^j(c+1)$.

Proof.

The proof trivially follows from Equation 4.30 since $u_k^{i,j}(c+1)$ is computed as the minimum of $u_k^{i,j}(c)$ and another term. □

Lemma 23 *Assume that in Algorithm 5:* $\forall q, 0 \leq q < d, \forall i, \forall j, 1 \leq j \leq S-q : Ub_{j,j+q}^i$ *is a valid upper bound to* $D_{j,j+q}^i$. *Then at step* j,k *of the algorithm, with* $k = j+d$, *in the iteration defined by Equations 4.29, 4.30 it holds:* $\forall c \geq 0, \forall i : u_k^{i,j}(c) \geq 0$.

Proof.

We prove the lemma by induction on c. The base case follows directly from the proof of Theorem 21, since the first two terms of Equation 4.27 are computed independently from the result of the iteration defined by Equations 4.29, 4.30.

For the induction step, assume that $\forall i : u_k^{i,j}(c) \geq 0$. We prove that $\forall i : u_k^{i,j}(c+1) \geq 0$. If according to Equation 4.30 , $u_k^{i,j}(c+1) = u_k^{i,j}(c)$, this is trivially true. Therefore, assume that $u_j^{i,j}(c+1) + \sum_{p=j}^{k-1} u_p^{i,j} = \bar{\alpha}_i\big(\Delta_{j,k}^{\max^U} - C + Ub_{j,k-1}^{(i)} + \sum_{p\in F, p\neq i} u_k^{p,j}(c)\big)$. Note that by the assumption on the correctness of Algorithm 5 up to step q and from Lemma 19 it follows: $\sum_{p=j}^{k-1} u_p^{i,j} \leq \bar{\alpha}_i\big(\Delta_{j,k}^{\max^U} - C + Ub_{j,k-1}^{(i)}\big)$. Note that since $\alpha_i(t)$ is an arrival curve, both $\alpha_i(t)$ and $\bar{\alpha}_i(t)$ are monotonically non-decreasing. Since furthermore by the induction hypothesis $\sum_{p\in F, p\neq i} u_k^{p,j}(c) \geq 0$, it holds: $u_j^{i,j}(c+1) + \sum_{p=j}^{k-1} u_p^{i,j} \geq \sum_{p=j}^{k-1} u_p^{i,j}$, which concludes the proof. □

Lemma 24 *Assume that the series defined by Equations 4.29, 4.30 converges to a fixed point* $\vec{u}_k^j$. *Then each* $u_k^{i,j}$ *value satisfies Equation 4.27.*

Proof.

Since $\bar{\alpha}_i(t)$ is monotonically non-decreasing and $u_k^{i,j}(c+1) \leq u_k^{i,j}(c)$ by Lemma 22, from Equation 4.30 it follows: $\bar{\alpha}_i\big(\Delta_{j,k}^{\max^U} - C + Ub_{j,k-1}^{(i)} + \sum_{p\in F, p\neq i} u_k^{p,j}(c+1)\big) - \sum_{p=j}^{k-1} u_p^{i,j} \leq \bar{\alpha}_i\big(\Delta_{j,k}^{\max^U} - C + Ub_{j,k-1}^{(i)} + \sum_{p\in F, p\neq i} u_k^{p,j}(c)\big) - \sum_{p=j}^{k-1} u_p^{i,j}$. Therefore, each $u_k^{i,j}(c+1)$ term can also be computed as:

$$u_k^{i,j}(c+1) = \min\Big(u_k^{i,j}(0), \bar{\alpha}_i\big(\Delta_{j,k}^{\max^U} - C + Ub_{j,k-1}^{(i)} + \sum_{p\in F, p\neq i} u_k^{p,j}(c)\big) - \sum_{p=j}^{k-1} u_p^{i,j}\Big). \tag{4.31}$$

Since $\vec{u}_k^j$ is a fixed point, it then follows:

$$u_k^{i,j} = \min\Big(u_k^{i,j}(0), \bar{\alpha}_i\big(\Delta_{j,k}^{\max^U} - C + Ub_{j,k-1}^{(i)} + \sum_{p\in F, p\neq i} u_k^{p,j}\big) - \sum_{p=j}^{k-1} u_p^{i,j}\Big) = \tag{4.32}$$

$$\min\Big(B_k^i, \min_{q:j+1\leq q\leq k}\big\{\bar{\alpha}_i\big(\Delta_{q,k}^{\max^U} - C + Ub_{q,k}^{(i)}\big) - \sum_{p=q}^{k-1} u_p^{i,j}\big\}, \bar{\alpha}_i\big(\Delta_{j,k}^{\max^U} - C + Ub_{j,k-1}^{(i)} + \sum_{p\in F, p\neq i} u_k^{p,j}\big) - \sum_{p=j}^{k-1} u_p^{i,j}\Big)$$

which is equivalent to Equation 4.27. □

Lemma 25 *The system of Equation 4.27 admits at most one solution.*

Proof.

By contradiction, assume that both $\vec{u}_k'^j$ and $\vec{u}_k''^j$ are different solutions to the system of Equation 4.27. Let $\mathbb{F}$ be the set of delay terms that have different values in $\vec{u}_k'^j, \vec{u}_k''^j$, e.g. $i \in \mathbb{F}$ iff $u_k'^{i,j} \neq u_k''^{i,j}$. Each delay term in $\mathbb{F}$ must be computed according to the third term in Equation 4.27, since in terms (1) and (2) $u_k^{i,j}$ is computed independently from all others $u_k^{p,j}, p \neq i$.

There are two possibilities: a) $\vec{u}_k'^j > \vec{u}_k''^j$ (or $\vec{u}_k''^j > \vec{u}_k'^j$); b) neither $\vec{u}_k'^j > \vec{u}_k''^j$ nor $\vec{u}_k'^j < \vec{u}_k''^j$ holds. In case a), since by definition of $\bar{\alpha}_i(t)$, each $u_k^{i,j}$ term must be maximal, it follows that $\vec{u}_k''^j$ (respectively, $\vec{u}_k'^j$) is not a solution to Equation 4.27.

In case b), without loss of generality assume that $\sum_{p\in F} u_k'^{p,j} \geq \sum_{p\in F} u_k''^{p,j}$. Since $\vec{u}_k'^j > \vec{u}_k''^j$ does not hold and the two solutions are different, then it must exist an element $i \in \mathbb{F}$ such that $u_k'^{i,j} < u_k''^{i,j}$. By definition of $\bar{\alpha}_i(t)$ and since both $\vec{u}_k'^j$ and $\vec{u}_k''^j$ are valid solutions it follows:

$$u_k'^{i,j} = \alpha_i\big(\Delta_{j,k}^{\max^U} - C + Ub_{j,k-1}^{(i)} + \sum_{p=j}^{k-1} u_p^{i,j} + \sum_{p\in F, p\neq i} u_k'^{p,j} + u_k'^{i,j}\big) - \sum_{p=j}^{k-1} u_p^{i,j} \tag{4.33}$$

and

$$u_k''^{i,j} = \alpha_i\big(\Delta_{j,k}^{\max^U} - C + Ub_{j,k-1}^{(i)} + \sum_{p=j}^{k-1} u_p^{i,j} + \sum_{p\in F, p\neq i} u_k''^{p,j} + u_k''^{i,j}\big) - \sum_{p=j}^{k-1} u_p^{i,j}. \tag{4.34}$$

Note that $\alpha_i(t)$ is monotonically non-decreasing and furthermore the argument of $\alpha_i(t)$ in Equation 4.33 is not smaller than the argument of $\alpha_i(t)$ in Equation 4.34 since by assumption $\sum_{p\in F, p\neq i} u_k'^{p,j} + u_k'^{i,j} \geq \sum_{p\in F, p\neq i} u_k''^{p,j} + u_k''^{i,j}$. Hence, it follows $u_k'^{i,j} \geq u_k''^{i,j}$, a contradiction. □

Theorem 26 *The series $\vec{u}_k^j(c)$ defined by Equations 4.29, 4.30 converges to $\vec{u}_k^j = (u_k^{1,j}, \ldots, u_k^{i,j}, \ldots)$, which is the unique solution to Equation 4.27.*

Proof.

Lemmas 22, 22 show that the series is monotonically non-increasing, and furthermore the series can not diverge.

Therefore, the series must must converge; let $\vec{u}_k^j$ be its fixed point. Since the series converges, Lemma 24 proves that $\vec{u}_k^j$ is indeed a solution to Equation 4.27. Finally, Lemma 25 proves that the system of Equation 4.27 can not admit more than one solution, hence $\vec{u}_k^j$ must be its unique solution. □

A final note is relative to computational complexity. Let K be the maximum number of iterations required by the series for convergence. Computing the value of any traffic delay curve $\bar{\alpha}_i(t)$ for a specific t has a maximum complexity of $O(S^2)$, the same as computing the curve. Computing the initial vector $(u_k^{1,j}(0), \ldots, u_k^{i,j}(0), \ldots)$ according to Equation 4.29 takes $O(\sharp FS^3)$; the traffic delay curve must be computed at most $O(\sharp FS)$ times, while $\Delta_{q,k}^{\max^U}, \sum_{p=q}^{k-1} u_p^{i,j}$ can be computed in constant time for each i, q by starting with $q = k$ and accumulating the values. Similarly, the whole iteration according to Equation 4.30 can be computed in $O(\sharp FKS^2)$; $\Delta_{j,k}^{\max^U}$ and $\sum_{p=j}^{k-1} u_p^{i,j}$ can be obtained from the corresponding terms computed in Equation 4.29 in constant time, and $\sum_{p\in F, p\neq i} u_k^{p,j}(c)$ can be computed in $O(\sharp F)$ at each step of the iteration using the same strategy as in Lines 7-10 of Algorithm 5. Since Algorithm 5 must solve Equation 4.27 $O(S^2)$ times, its overall computation complexity is $O\big(\sharp FS^4 \max(S, K)\big)$. Note that for general traffic delay curves, the series might converge in an infinite number of steps. In practice, as we show in Section 4.4.4, the iteration typically converges quickly in a limited number of steps. Furthermore, following the proof of Theorem 26, it can be shown that $\forall c, i : u_k^{i,j}(c) \geq u_k^{i,j}$. Therefore, it is possible to halt the recurrence after a fixed number of steps and still obtain a valid upper delay bound.

Bound Tightness

It remains to discuss whether the $\{Ub_{j,k}\}$ bounds returned by Algorithm 5 are tight or not, e.g. if $Ub_{j,k} = D_{j,k}$ holds for all possible cache profiles and flows. For a system with a single flow α_i, in [70] we show that the obtained $Ub_{j,k}^i$ bounds are tight if $\alpha_i(t)$ is a concave function. Unfortunately, in a system with multiple flows, the bounds are not tight anymore. This is because each $Ub_{j,k}^i$ value is maximized independently of the delay bound $Ub_{q,k}^{(i)}$ for all other flows that is used in the second term of Equation 4.27. Assume that $u_k^{i,j}$ is equal to the second term of Equation 4.27 for a specific value of q. Then $Ub_{j,k}^i$ is computed based on the assumption that traffic from all other flows cause maximum interference in subinterval $\{s_q, \ldots, s_k\}$. However, in the pattern of cache interference that results in the actual worst case delay $D_{j,k}$, the interference of flows other than α_i on $\{s_q, \ldots, s_k\}$ could be less than $Ub_{q,k}^{(i)}$, in particular because they could cause more interference on $\{s_j, \ldots, s_{q-1}\}$. Therefore, the computed $Ub_{j,k}^i$ would be larger than the real worse case.

A clarifying example is shown in Figure 4.13 for a system with two flows α_1, α_2 and RR arbitration with $L_i = C_i = 1$. The task under analysis has three scheduling intervals $\{s_1, s_2, s_3\}$ with $exec_1^U = exec_3^U = 2$, $\mu_1^{\max} = \mu_3^{\max} = 2$ and $exec_2^U = 3, \mu_2^{\max} = 0$. Figure 4.13 shows the pattern of memory requests that results in the maximum delay $D_{1,3} = 6$ for the task under analysis. Note that in the figure, flow α_2 can not delay the task under analysis in $[12, 13]$ because $\alpha_2(13 - 9) = 1$, e.g. the flow does not have enough traffic to cause a delay of 2 time units in s_3. However, running Algorithm 5 returns a bound $Ub_{1,3} = 7$. This is because when computing $u_k^{2,j} = u_3^{2,1}$ in Equation

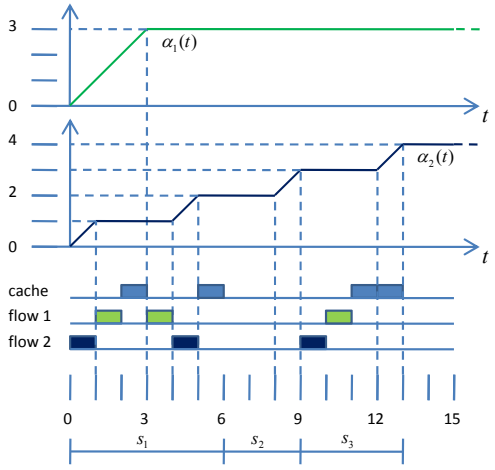


Figure 4.13: Example of non-tight bound.

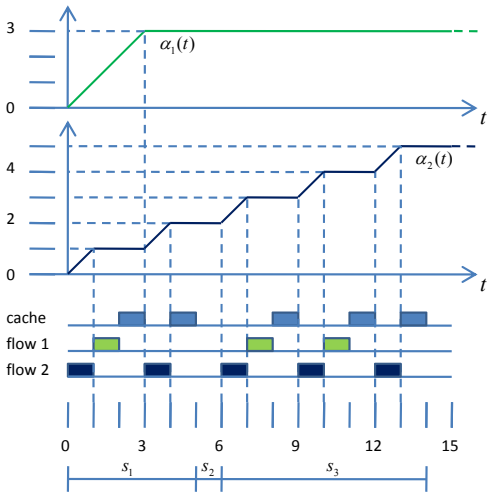


Figure 4.14: Example complex delay pattern.

4.27 at step $j = 1, k = 3$, for $q = 3$ the algorithm uses a value $Ub^{(2)}_{q,k} = Ub^1_{3,3} = 2$, while in Figure 4.13 flow α_1 delays the task under analysis for one time unit ($Ub^1_{3,3}$ is a bound on the delay for scheduling interval s_3 only, hence flow α_1 has enough traffic to delay both memory requests of the task under analysis). As a consequence, in the algorithm scheduling interval s_3 is delayed more than in the real worst case, allowing an additional request of flow α_2 to delay the task under analysis.

Whether it is possible to obtain tight bounds, at least for concave arrival curves, in polynomial time in the number of scheduling intervals of the task under analysis is left as an open question. Consider the scenario in Figure 4.14, where we use the same α_1 function as in Figure 4.13 but different α_2 and cache profile, with $exec^U_1 = 0, \mu^{\max}_1 = 2$, $exec^U_2 = 1, \mu^{\max}_2 = 0$, $exec^U_3 = 0, \mu^{\max}_3 = 3$. It is easy to see that to obtain the maximum delay $D_{1,3}$, flow α_1 must now delay scheduling interval s_1 for one and s_3 for two time units; otherwise, flow α_2 could not generate the required three units of traffic in s_3. This example shows that it can be difficult to predict which subinterval should suffer maximum interference for each flow. In fact, we strongly suspect that if a tight bound can be obtained, then an exponential number of scheduling interval intervals would need to be examined. We plan to study this case in more details as part of our future work.

Backlog computation for buffered flows

An upper bound on the backlog b_i for each buffered flow α^*_i can be easily obtained using the theory of Network Calculus [16]. Let β_i be a strict service curve for α^*_i, e.g. in any interval of length t in which the flow is backlogged, $\beta_i(t)$ is a lower bound to the amount of service provided by main memory to the flow. Then:

$$b_i \leq \sup_{t \geq 0} \{\alpha^*_i(t) - \beta_i(t)\}. \tag{4.35}$$

```
cpuid; //synchronization barrier
mov ECX, 0000 0000H;
rdpmc; //read Counter 0
//move value from DL:EAX to reservation controller
mov [RESCON_COUNTER0_H], DL;
mov [RESCON_COUNTER0_L], EAX;
mov ECX, 0000 0001H;
rdpmc; //read Counter 1
mov [RESCON_COUNTER1_H], DL;
mov [RESCON_COUNTER1_L], EAX;
```

Figure 4.15: Checkpoint Assembler Code.

It remains to compute β_i. Given our assumptions, it can be shown that β_i is minimized when flow α_i^* has lowest static priority. Let $\alpha(t)$ be the arrival curve for the processing core where the task under analysis is executed. It follows:

$$\beta(t) \geq t - \sum_{\alpha_i \in F} \alpha_i(t) - \sum_{\alpha_i^* \in F} \alpha_i^*(t) - \alpha(t). \tag{4.36}$$

Handling Write Buffers

Many modern architectures employ a *write buffer* for the last cache level: in this case, write requests are not immediately executed, but rather temporary stored in the write buffer. The write buffer then takes care of writing the dirty cache line to main memory when there is time available, e.g. the write buffer produces requests with lower priority than cache fetches. Write buffers can be included in the analysis by modeling them as additional buffered flows. Since replacements of dirty cache lines do not generate immediate write-back, they should not be accounted for in $\mu_{i,j}^{\max}, \mu_{i,j}^{\min}$. Instead, we define new values $\tilde{\mu}_{i,j}^{\max}, \tilde{\mu}_{i,j}^{\min}$ to be the maximum and minimum number of replaced cache lines in scheduling interval $s_{i,j}$ or equivalently, the number of cache lines pushed to the write buffer. A buffered arrival curve $\tilde{\alpha}_i^*$ can then be computed for the write buffer of processing core PE_i using the methodology described in Section 4.4.1 with the $\tilde{\mu}_{i,j}^{\max}, \tilde{\mu}_{i,j}^{\min}$ values. The analysis of Section 4.4.2 is applied including buffered flows for all write buffers and assigning them lowest static priority. Note that the analysis must include a flow for the write buffer of the processing core where the task under analysis is executed, since it can interfere with cache fetches; however, since we know that the task under analysis is running, the arrival curve for the write buffer can be obtained assuming that no other task runs on the processing core.

4.4.3 Cache Profile Measurement

We devised a testing methodology to experimentally obtain the cache profile in the interval-based model; in particular, we detail how the execution time and number of cache misses can be accurately measured for each scheduling interval. Our implementation uses the Intel Core Microarchitecture architectural performance counters [47], but other CPU architectures such as IBM PowerPC provide similar support for CPU self-measures. Support was added by modifying the Linux/RK kernel [67]. The Core Microarchitecture specifies support for three architectural performance counters, each of which can be configured to count a variety of internal events. In particular, we used Counter 0 to count the number of elapsed CPU clock cycles and Counter 1 to count the number of level-2 cache misses. To accurately measure task execution without the effects of OS overhead, we configured both counters to be active only when the CPU is executing in user mode. Finally, we allowed reading the counter values from user mode with the `rdpmc` instruction (the counters can still be written and configured only in kernel mode) to reduce measurement overhead.

Counters are read inside each task by adding the checkpoint code in Figure 4.15 at the end of each scheduling interval. The `cpuid` instruction inserts a synchronization barrier, i.e. it makes sure that all instructions fetched before `cpuid` are completed before the counters are read; this is required to cope with out-of-order execution. The counter values are then sent through the PCI bus to an external data acquisition peripheral, based on the peripheral scheduler detailed in Section 2.1; this ensures that no write to system memory is performed at the checkpoint, which could cause an additional cache miss. The peripheral scheduler determines the execution time and number of cache misses in each scheduling interval computing the difference between the values obtained in successive checkpoints. After the task has finished, the computed values are read back from the peripheral scheduler. Worst-case and best-case execution time and number of cache misses can be determined as the maximum and minimum, respectively, over several task runs. Note that performance counters are not task-specific, so we had to modify the kernel to support reading the counters in a multitasking environment. We added two new fields to the task descriptor, `counter_extime` and `counter_cmisses`, to store the counter information. When a task is created, the fields are set to zero. When a task is preempted, the kernel first reads the counter values and saves them in the preempted task's descriptor. Then, it writes the values in the preempting task's descriptor back in the counters. Finally, the kernel writes the id of the preempting task in a register of the reservation controller, so that the controller can correctly associate the received information with the running task.

We implemented a compiler pass using the LLVM compiler infrastructure [52] to automatically add checkpoint code to the task. In the current implementation, the designer must manually identify the scheduling interval boundaries selecting an initial and final basic block for each scheduling interval. The choice involves a tradeoff, as smaller scheduling intervals provide better information and tighter WCET bounds but at the price of increased measurement overhead.

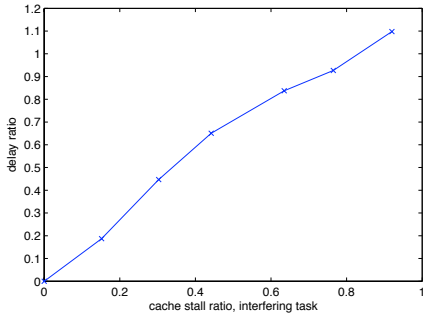


Figure 4.16: Measured computation time increase with two cores.

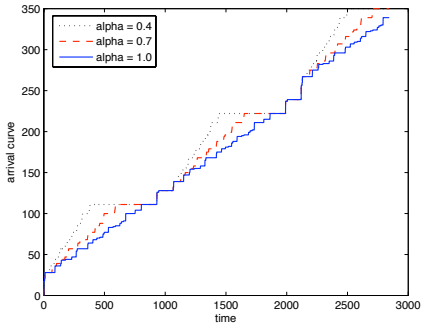


Figure 4.17: Computed arrival curves for different α values.

4.4.4 Experimental Results and Simulations

We performed experiments on an Intel Core microarchitecture platform to understand the severity of the memory interference problem. Using a PC COTS platform allowed us to access PCIe slots and easily measure task performance using Intel performance counters; however, to obtain meaningful measures we slowed down the Front Side Bus obtaining a speed of 1Ghz for each core and a theoretical bandwidth of 2.4GB/s, which is in line with typical speeds for embedded systems.

We selected a CoreQuad CPU implementing four cores on two joined CPU dies. The two cores on each die share level 2 cache; hence, to mirror our described model we disabled cores PE_2 and PE_4 and used only one core on each CPU die. We engineered a software task that continuously suffers cache misses in order to measure an upper bound to the delay suffered due to memory interference. The task allocates a buffer with size double its level 2 cache, and then cyclically reads one word at a time from each cache line. This forces the task to suffer a cache miss for every read operation; we measured both the total duration of each task instance and the number of cache misses, and found a cache stall time (defined as the ratio $\frac{\mu_{i,j}^{\max} C}{exec_{i,j}^{U} + \mu_{i,j}^{\max} C}$, e.g. the percentage of time spent by the time stalling with no external interference) of 92%. We then modified the task by inserting a variable number of *nop* instructions between successive reads. In this way, we obtained versions of the task with stall time of around 15%, 30%, 45%, 60% and 75%. We ran one copy of the task with 92% stall time on core PE_1 and one copy of the task with variable stall time on PE_3, and measured the increase in computation time.

Results are reported in Figure 4.16 as the ratio between the maximum measured delay and the computation time of the task when run in isolation, obtained over 4 runs. Note that the delay ratio for the first task increases almost linearly with the cache stall time of the second task, peaking at a 1.1 when both tasks saturate memory with cache fetch requests. We believe that the increase in computation time is over two times due to additional arbitration overhead; in particular, our platform uses DDRAM for main memory, whose response time is highly dependent on data location. To build a safe upper bound, a safe estimate for the time C required to service each memory request must be used, but

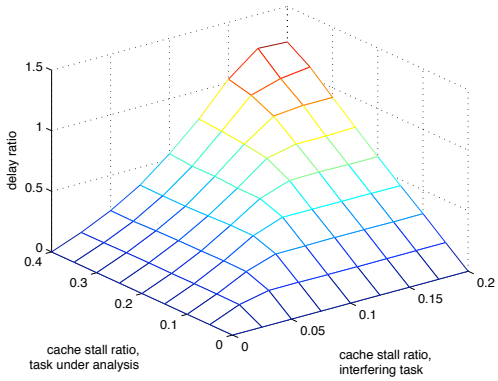


Figure 4.18: Delay ratio, $\sigma = 0.2, \alpha = 0.8$, 4 cores.

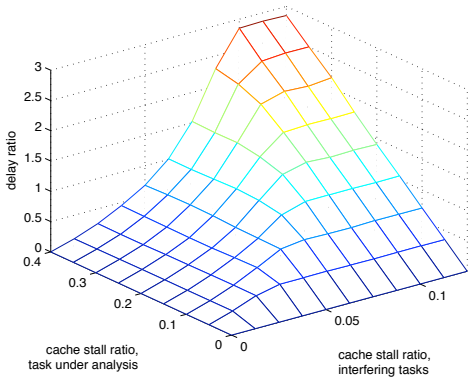


Figure 4.19: Delay ratio, $\sigma = 0.2, \alpha = 0.8$, 8 cores.

in practice, when the task is run in isolation each subsequent cache fetch can often be completed in less time, leading to a decreased computation time. In this sense, it is important to note that the results reported in Figure 4.16 are average case delays, since exactly synchronizing the tasks to obtain the worst case interference as described in Lemma 16 is impossible. Finally, we repeated the experiment after adding to our testbed a custom-designed PCIe peripheral that continuously sends write requests to main memory. Our developed 8-lanes PCIe peripheral has a theoretical throughput of 2GB/s and can therefore almost saturate the available memory bandwidth. When running both tasks at 92% stall time, we measured a delay ratio of 1.96, e.g. the computation time of the measured task increased 2.96 times.

Unfortunately, the obtained experimental results can not be directly compared to our analysis bounds because we do not know the exact L_i values nor the arbitration scheme for the employed platform. This is typical in the PC market, where vendors are wary of revealing full implementation details. However, as we discussed in Section 4.1 such information is usually available for embedded system platforms.

We also performed extensive simulations to understand how the delay bound varies as a function of task parameters and how fast the delay iteration of Section 4.4.2 converges. In particular, we decided to simulate a N-core system with RR arbitration, $C = L$ and one task for each core with $S = 10$ scheduling intervals. Tasks are synthetically generated according to three parameters σ, β and α. For each task and scheduling interval, we first generate $exec_{i,j}^{U}$ according to a uniform distribution with mean $100C$ and coefficient of variation σ. We then generate a cache stall ratio for the scheduling interval according to a uniform distribution with mean β and coefficient of variation σ and compute $\mu_{i,j}^{\max}$ accordingly. Finally, we set $exec_{i,j}^{L} = \alpha\ exec_{i,j}^{U}, \mu_{i,j}^{\min} = \alpha\ \mu_{i,j}^{\max}$ and we set the period to $p_i = \Delta_{1,S_i}^{\max^U}$.

Figure 4.17 shows computed $\alpha_i(t)$ functions for a synthetic task τ_i with $\sigma = 0.2, \beta = 0.1$ over the interval $[0, 3p_i]$. Three different $\alpha_i(t)$ functions are obtained by setting α equal to $0.4, 0.7$ and 1.0 for the task. Note that for smaller values of α, the arrival curve becomes larger. This is because the time window computation in Section 4.4.1 assumes that the maximum amount of memory requests can happen together with the lowest execution time for each scheduling

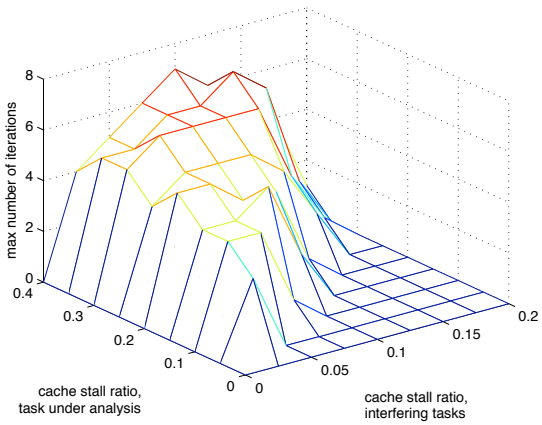


Figure 4.20: Max number of iterations, $\sigma = 0.2, \alpha = 0.8$, 4 cores.

interval; hence, the same amount of requests can arrive in a smaller window since $exec^L_{i,j}$ decreases with α. However, all arrival curves eventually assume the same value at the end of each period since the time window is constrained by the gap g between the last scheduling interval and the next period.

Figure 4.18 shows simulation results in terms of the delay ratio between the computed upper delay bound Ub_{1,S_i} and the WCET $\Delta^{\max^U}_{1,S_i}$ for the task executed on the first core in a system with $N = 4$ cores. In the figure, we fix $\alpha = 0.8, \sigma = 0.2$ and vary the cache stall parameter β in $[0, 0.4]$ for the task under analysis and in $[0, 0.2]$ for the other three interfering tasks. Each point in the graph is computed as the average over 100 runs; each run, which involves generating the tasks, computing arrival curves and applying Algorithm 5 took less than a second on a modern PC.

Note that the delay ratio increases almost linearly with the stall ratio for the interfering tasks, until it saturates at roughly three times the stall ratio for the task under analysis (for example, for $\beta = 0.4$ the graph saturates at a value slightly higher than 1.2). This is expected: at a certain point, the memory traffic generated by each interfering core becomes so high that the delay computed by Algorithm 5 is dominated by the blocking factor B^i_j, which does not depend on flow traffic. Also note that a stall ratio of 0.2 for the interfering tasks is enough to cause delay saturation for a task under analysis with stall ratio of 0.4. This is mainly because Algorithm 5 must take into account the mutual effect of multiple flows; the delay caused by one flow can "stretch" a scheduling interval allowing more interfering traffic from other flows.

Figure 4.19 shows a similar simulation for a system with $N = 8$ cores and the same $\alpha = 0.8, \sigma = 0.2$ parameters. Note that the graph is very similar to the one in Figure 4.18, except that it saturates at roughly seven times the stall ratio for the task under analysis, and saturation is reached for a stall ratio of 0.1 for the intefering tasks. Finally, while we performed additional simulations by varying the values of α and σ, their results are very similar to the one showed in Figures 4.18, 4.19. This is mainly because as shown in Figure 4.17, independently of the value of β all arrival curves reach the same value at the end of a period. As a consequence, the graphs saturates at roughly the same stall ratio for

the interfering tasks.

Finally, in Figure 4.20 we plot the maximum number of steps required for convergence by the series defined by Equations 4.29, 4.30, given the same scenario as in Figure 4.18. Note that the series converged in at most 7 steps in all simulations. This is because the delay functions $\bar{\alpha}_i(t)$ obtained from the arrival curves computed in Section 4.4.1 resemble step functions; when step functions are used in the series, at least one element $u_k^{i,j}(c+1)$ decreases by the size of one "step" at each iteration; this in turn causes quick convergence. As expected, the number of required steps is larger when the difference between the cache stall ratio for the task under analysis and the cache stall ratio for the interfering tasks is large, since in this case the computed delay bound tends to be dominated by traffic delay curves $\bar{\alpha}_i(t)$.

4.5 Related Work

Apart from our work, several papers address the problem of interference at the main memory level. A closely related work is described in [87], where the impact of PCI bus load on the computation time of a single task is investigated. An empirical approach is introduced that estimates the overall delay incurred by a task based on measures on reference tasks; no exact worst case pattern is derived. On the contrary, the focus in our work is on the design of an analysis architecture that is able to provide provable worst case bounds. An analytical approach to describe the impact of I/O load on real-time tasks is introduced in [44]. However, the analysis is restricted to a single DMA controller using a highly predictable cycle-stealing arbitration, and can not be applied to commodity systems. Furthermore, no explicit bound on peripheral load is considered, possibly resulting in WCET overestimation. Schliecker et al. [86, 85] have proposed an event-based model to estimate delay for both computation and communication activities in a multicore system-on-chip. This methodology takes into account the effect of interference for access to shared memory, similarly to our work, but is based on a detailed model of interdependency among cache misses that can be difficult to derive.

Closely related to timing analysis for systems with shared resources, there are also works that focus on spatial interference or data conflicts caused due to cache accesses, e.g., Guan et al. [41] and Li et al. [55]. In [41], Guan et al. propose a scheduling algorithm and a schedulability test based on linear programming for multicore systems with shared L2 cache. Timing analysis, based on linear programming, considering cache conflicts is provided in [55].

Approaches like Modular Performance Analysis [102], or SymTA/S [97] cannot be applied to analyze the worst-case response time for systems with unbuffered resource accesses, due to their blocking behavior. These methods are useful for analyzing the required buffer size and the events that are produced by the resource for systems with asynchronous resource access.

Another research direction is to eliminate the interferences by applying more predictable resource arbiters. By applying static analysis to compute the feasible traces, Rosen et al. [82] developed approaches to derive efficient TDMA arbitration policies. Schranzhofer et al. [88] developed a framework for analyzing the worst-case response time of real-time tasks when TDMA is applied to arbitrate access to a shared resource. However, many available

COTS arbiters do not support TDMA arbitration.

4.6 Future Work

While our multicore delay analysis in Section 4.4 can already compute upper delay bounds for a variety of arbitration schemes, it could be further expanded to cover more general assumptions. First of all, the arrival curve derivation in Section 4.4.1 assumes a static schedule. We plan to investigate how to extend the derivation to on-line real-time schedulers such as RM or EDF. Second, the analysis makes simplifying assumptions about the memory controller: fetch times in DRAM are not constant, but depend on the history of previous accesses. Since we do not consider past history, the only safe assumption is to use the worst case access time in DRAM, which can be very pessimistic. Furthermore, modern DRAM controllers can access multiple memory banks in parallel, possibly reducing contention if data is suitably arranged in main memory.

Finally, the analysis assumes private last-level caches. The current trend in multicore design is towards the adoption of large shared level-3 caches. Therefore, it would be interesting to extend the analysis to cover multiple levels of shared memory elements. Unfortunately, a basic understanding of the principles behind our delay analysis clearly reveals that the addition of multiple levels of shared resources introduces additional sources of contention in the system. Hence, analysis bounds would be even more pessimistic compared to the average case. In fact, even without the addition of shared level-3 cache, the evaluation of Section 4.4.4 shows that analysis bounds scale poorly with increasing number of cores. As such, we believe that for systems comprising four or more cores, some engineering mechanisms must be applied to reduce the effect of memory contention. We will detail our solution in the following chapter.

Chapter 5

Predictable Task Execution

As shown in Chapter 4, computing precise bounds on timing delays due to contention for access to shared resources is difficult. Even though our analysis produces safe upper bounds, the worst case access behavior tends to be very pessimistic compared to the average case due to the unpredictable behavior of arbiters of physically shared COTS resources (like caches, memories, and buses). Section 4.4.4 shows that the computation time of a task can increase linearly with the number of suffered cache misses due to contention for access to main memory. In a system with three active components, a task's worst case computation time can nearly triple. To exploit the high average performance of COTS components without experiencing the long delays occasionally suffered by real-time tasks, we need to control the operating point of each COTS shared resource and maintain it below saturation limits.

This chapter shows that this is indeed possible by exploiting control abstractions and enforcing a high-level coscheduling mechanism among active COTS components. We propose two solutions for PC architectures, based on the level of control possible in the system. For systems where peripherals are put under the control of the I/O management system of Section 2.1, but cache misses produced by CPU tasks are still inherently unpredictable, we propose an on-line optimization heuristic. The heuristic optimistically assigns a computation time budget to each task assuming that it suffers no memory interference. At run-time, the task progress is then monitored and peripheral traffic is allowed if we can ensure that the task will still meet its deadline. Unfortunately, our on-line heuristic does not solve the core of the problem, in the sense that cache fetches and peripheral traffic can still interfere in main memory. Hence, no worst-case guarantee can be provided to I/O flows.

To solve this issue, in Section 5.2 we propose our PRedictable Execution Model (PREM). PREM uses a novel program execution model with three main features: **(1)** jobs are divided into a sequence of non-preemptive scheduling intervals, as in Section 4.1; **(2)** some of these scheduling intervals (named **predictable intervals**) are executed *predictably* and *without cache-misses* by prefetching all required data at the beginning of the interval itself; **(3)** the execution time of **predictable intervals** is kept constant by monitoring CPU time counters at run-time. Using PREM, we can then enforce a coscheduling mechanism that serializes arbitration requests of all active components (CPU cores

and I/O peripherals). During the execution of a task's predictable interval, a scheduled peripheral can access the bus and memory without experiencing delays due to cache misses caused by the task's execution. In other words, PREM can be seen as a *software control abstraction*, able to control task accesses in main memory, in the same way that the real-time bridges described in Section 2.1 control peripheral accesses.

PREM can be used with a high level programming language like C by setting some programming guidelines and by using a modified compiler to generate predictable executables. The programmer provides some information, like beginning and end of each predictable execution interval, and the compiler generates programs which perform cache prefetching and enforce a constant execution time in each predictable interval. In light of the above discussion, we argue that real-time embedded applications should be compiled according to a new set of rules dictated by PREM. At the price of minor additional work by the programmer, the generated executable becomes far more predictable than state-of-the-art compiled code, and when run with the rest of the PREM system, shows significantly reduced worst-case execution time.

5.1 Coscheduling of Tasks and Peripherals

It is important to note that even when the delay bound computed in Chapter 4 is tight, it is rare that at run-time a task will suffer a delay equal to the bound: a particular pattern for both cache misses and peripheral transactions is required to produce the worst case. As such, accounting for the worst case delay inflicted by all peripherals and other cores in the execution time budget of each task can lead to a large waste of resources. For single core systems, we propose an alternative solution based on a run-time adaptive algorithm. Consider a task under analysis composed by S scheduling intervals $\{s_1, \ldots, s_S\}$, each with maximum execution time $\{e_1, \ldots, e_S\}$. e_i is computed taking into account the time required for all memory operations in scheduling interval s_i, but assuming no interference in main memory; based on the notation in Section 4.4, if each memory operation takes C time units to complete, then $e_i = exec_i^U + C\mu_i^{\max}$. Furthermore, we assume that all peripherals in the system are under the control of an I/O management system such as the one described in Section 2.1. The idea is to assign to a task the minimal time budget of $\sum_{1 \leq i \leq S} e_i$, and then to monitor the actual execution of the task and open the real-time bridges whenever possible. At run-time, information on the execution time consumed by the current job is sent to the peripheral scheduler at each checkpoint. The peripheral scheduler uses this information to determine the actual execution time $\hat{e}_i$ of scheduling interval s_i for the current job. The *accumulated slack* time after scheduling interval s_i can then be computed as $\sum_{1 \leq j \leq i}(e_j - \hat{e}_j)$; the slack time is the maximum delay that the task can suffer while still meeting its execution time budget. We can then design a coscheduling algorithm that strives to maximize the amount of time that the real-time bridges are opened under the constraint that the slack can never become negative. Our proposal is to integrate both the analysis and coscheduling techniques in a mixed-criticality system. Inspired by the avionic domain, we consider two types of guaranteed real-time tasks: *safety critical* tasks like flying control, that have stringent delay and verification requirements, and *mission critical* tasks that are still hard real-time but have lower criticality. We propose to schedule safety critical tasks blocking

Algorithm 6 Adaptive Algorithm

```
procedure JOBSTART
    slack := 0
    i := 0
    CloseBridge()
end procedure
procedure CHECKPOINT(ê)
    i := i + 1
    slack := slack + e_i − ê
    if D(s_{i+1}) ≤ slack then
        OpenBridge()
    else
        CloseBridge()
    end if
end procedure
```

all I/O traffic except the one from peripherals used by the task, and to account the delay in the time budget. For *mission critical* tasks we instead use coscheduling. Finally, we also assume that the system runs best effort or soft real-time tasks for which all real-time bridges are opened.

Algorithm 6 is our main coscheduling heuristic. For simplicity, we describe the algorithm for a single controlled task and a single peripheral in a single core system, but it can be easily extended to a multitasking environment with multiple real-time bridges. In the algorithm, $D(s_i)$ returns the maximum delay suffered by the task under analysis in scheduling interval s_i, computed using any of the applicable algorithms in Chapter 4. Communication from the task to the peripheral scheduler triggers the algorithm at the beginning of each job and at each checkpoint. The algorithm maintains two variables: `i` is the index of the last executed scheduling interval, and `slack` represents the accumulated slack. At the end of each scheduling interval s_i, the algorithm first recomputes the slack and then performs a check: if the slack is at least equal to the maximum delay $D(s_{i+1})$, then the real-time bridge is opened because we are sure that the slack will be non negative after the next scheduling interval s_{i+1} is executed. Otherwise, the real-time bridge is kept closed.

The limitation of Algorithm 6 is that it greedily "allocates" all slack to the next scheduling interval by immediately opening the real-time bridge. This can lead to a suboptimal allocation, as scheduling interval s_{i+1} could be short and have a lot of cache misses while scheduling interval s_{i+2} could be longer with very few cache misses. If we have additional information on the task, we can potentially do better using a predictive heuristic. In particular, Algorithm 7 assumes that the average case execution time avg_i and average case delay $D^{avg}(s_i)$ for each scheduling interval s_i is known. The algorithm keeps track of the *predicted slack*, i.e. the total slack assuming that all future scheduling

Algorithm 7 Predictive Algorithm

procedure JOBSTART

$slack := 0$

$pslack := \sum_{k=1}^{S}(e_k - avg_k)$

$i := 0$

CloseBridge()

end procedure

procedure CHECKPOINT($\hat{e}$)

$i := i + 1$

$slack := slack + e_i - \hat{e}$

$tmp := pslack := pslack + avg_i - \hat{e}$

for all k in ORDERED_LIST ($\frac{avg_j + D^{avg}(s_j)}{D^{avg}(s_j)}$) **do**

if $k > i + 1 \wedge D^{avg}(s_k) \leq tmp$ **then**

$tmp := tmp - D^{avg}(s_k)$

end if

if $k == i + 1$ **then**

if $D(s_{i+1}) \leq slack \wedge D^{avg}(s_{i+1}) \leq tmp$ **then**

OpenBridge()

else

CloseBridge()

end if

return

end if

end for

end procedure

intervals will execute for $\hat{e}_j = avg_j$. We can then compute a strategy that maximizes the amount of time that the real-time bridge is opened by allocating the predicted slack among all future scheduling intervals: if we decide to open the real-time bridge during s_j, we consume an amount of slack equal to $D^{avg}(s_j)$ and the real-time bridge is opened for $avg_j + D^{avg}(s_j)$ time units. It is easy to see that this allocation problem is equivalent to the KNAPSACK problem [49], which is well known to be NP-hard. We therefore use a sub-optimal polynomial time greedy solver: off-line, we order all scheduling intervals by non-increasing values of $\frac{avg_j + D^{avg}(s_j)}{D^{avg}(s_j)}$. At run-time, we perform the allocation by iterating through the list ignoring all scheduling intervals already executed. When the iteration arrives to the next scheduling interval s_{i+1}, the real-time bridge is opened if the remaining predicted slack is greater or equal than $D^{avg}(s_{i+1})$.

SLACKONLY	ADAPTIVE	PREDICTIVE	BOUND
4.89%	31.21%	36.65%	40.85%

Table 5.1: Benchmark Results.

Note that Algorithm 7 is not the only possible predictive algorithm; in fact, no on-line algorithm can be optimal, since any optimal algorithm must known exactly the execution time of future scheduling intervals, i.e. it must be clairvoyant. However, for the sake of comparison it is interesting to compute an upper bound on the best possible performance of any on-line predictive algorithm. Assume that for a specific run, $\hat{e}_i$ is the execution time of s_i assuming that the real-time bridge is closed, and $\bar{e}_i$ is the execution time assuming that the real-time bridge is opened. An upper bound can be computed by solving the following integer linear programming problem:

$$\max \sum_{i=1}^{S} x_i \bar{e}_i \tag{5.1}$$

$$\forall i, 1 \leq i \leq S : x_i D(s_i) \leq \sum_{j=1}^{i-1} (e_j - (1 - x_j)\hat{e}_j - x_j \bar{e}_j) \tag{5.2}$$

$$\forall i, 1 \leq i \leq S : x_i \in \{0, 1\}, \tag{5.3}$$

where $\{x_1, \ldots, x_S\}$ are indicator variables (i.e., $x_i = 1$ if the real-time bridge is opened during s_i). Equation 5.1 maximizes the open time, while Equation 5.2 expresses the slack constraint.

5.1.1 Experimental Results

To validate our architecture, we performed experiments on a COTS PC platform comprised of an Intel Core2 CPU and an Intel 975X system controller. Similarly to Section 4.4.4, to derive meaningful measures we changed the Front Side Bus clock frequency obtaining a speed of 900Mhz for the CPU and a theoretical bandwidth of 2.4Gbyte/s for main memory, which is in line with typical values for embedded platforms.

We evaluated the performance of the described coscheduling algorithms on our platform using a MPEG decoder [36] benchmark. We chose MPEG for two main reasons: it is both a memory and I/O intensive application, and it is representative of the type of video computation that is becoming increasingly important for mission control in avionic systems. We collected average and worst case statistics on a test video clip after placing multiple checkpoints for each frame; the MPEG decoder is run as a periodic task, with 20 scheduling intervals in each period. To mirror the behavior of a real application and increase the number of cache misses, we also ran a higher priority task that preempts the MPEG decoder every 1ms and replaces its cache content. Results averaged over 50 runs are shown in Table 5.1 in term of the percentage of time that the real-time bridge is opened in the task period. In the table, SLACKONLY represents a baseline solution where the real-time bridge is kept closed while the task is executing and is opened after the task has finished for its remaining time budget $\sum_{1 \leq i \leq S}(e_i - \hat{e}_i)$; ADAPTIVE is Algorithm 6; PREDICTIVE

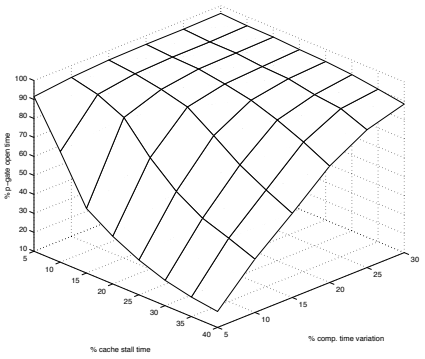


Figure 5.1: Synthetic Tasks, ADAPTIVE, $\sigma = 0.1$

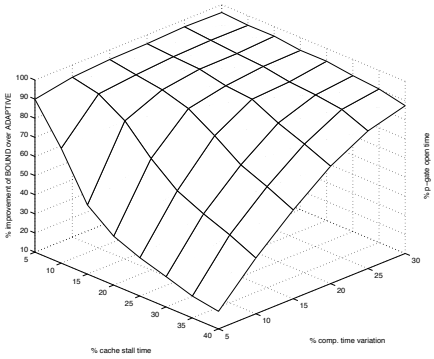


Figure 5.2: Synthetic Tasks, ADAPTIVE, $\sigma = 0.2$

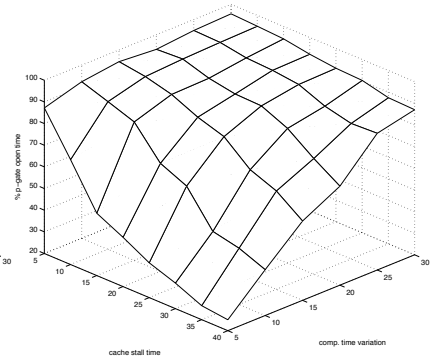


Figure 5.3: Synthetic Tasks, ADAPTIVE, $\sigma = 0.4$

is Algorithm 7; BOUND is the bound computed by solving the ILP problem of Equations 5.1-5.3. Note that since BOUND is not implementable at run-time, we computed the bound offline using measured values of execution times and number of cache misses. We can see that SLACKONLY tends to perform very poorly; ADAPTIVE is within 30% of BOUND and PREDICTIVE is roughly in between the two, which seems to suggest that prediction offers limited improvement.

To check whether the obtained results hold for more general settings, we also performed extensive simulations on synthetic tasks, each composed of 20 scheduling intervals, varying three parameters σ, α, β. For each task and each scheduling interval s_i, we first generated the average execution time avg_i from a uniform distribution with constant mean and coefficient of variation σ, and the average cache stall time $stall_i$ from a uniform distribution with mean β and coefficient of variation σ. We then simulated 10 task runs by extracting for each run and each scheduling interval an execution time $\hat{e}_i = avg_i(1 + \bar{\alpha})$ and a number of cache misses equal to $stall_i(1 + \bar{\alpha})$, where $\bar{\alpha}$ is extracted from a uniform distribution with mean 0 and maximum value α. Note that this implies $wcet_i = avg_i(1 + \alpha)$, i.e. α is the increase in execution time between the average and the worst case.

We focus on results for the ADAPTIVE and BOUND cases: Figures 5.1, 5.2, 5.3 show the value of ADAPTIVE (percentage of time that the real-time bridge is opened) for values of σ equal to $0.1, 0.2, 0.4$ respectively. Figures 5.4, 5.5, 5.6 shows the competitive ratio of $\frac{\text{BOUND}-\text{ADAPTIVE}}{\text{ADAPTIVE}}$, again for $\sigma = 0.1, 0.2, 0.4$. All points are averages over 10 tasks (100 runs total of the simulator). We varied the average cache stall time β between $[0.05, 0.4]$ and the execution time variation α between $[0.05, 0.3]$; axis direction is inverted between the two sets of figures for easier visualization. Note that the definition of α implies SLACKONLY=$\frac{\alpha}{1+\alpha}$ for all cases. First note that the obtained results are very close for the different values of σ, which seems to indicate that none of the tested algorithm is very sensitive to variations in the size of scheduling intervals. The second main observation is that the performance of the algorithms depends on the difference between α and β. For $\alpha > \beta$, both algorithms can open the real-time bridge almost all the time because the high wcet variability forces us to over-provision the execution time budget of the task;

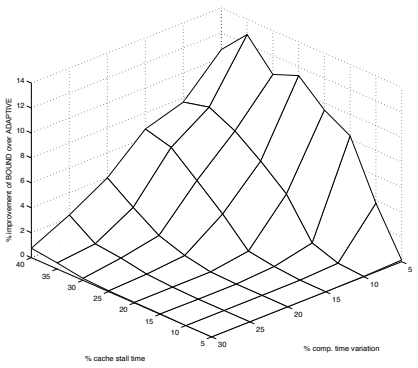


Figure 5.4: Synthetic Tasks, algorithm ratio, $\sigma = 0.1$

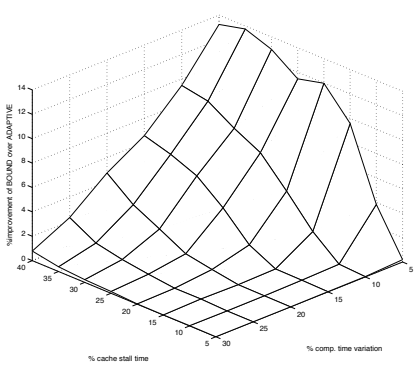


Figure 5.5: Synthetic Tasks, algorithm ratio, $\sigma = 0.2$

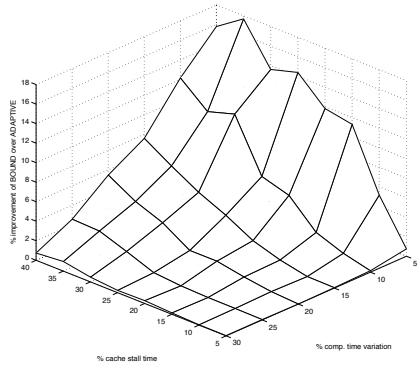


Figure 5.6: Synthetic Tasks, algorithm ratio, $\sigma = 0.4$

however, note that a coscheduling algorithm is still needed to guarantee safety, as there are times where the real-time bridge must be closed to ensure that the task meets its deadline. In the case $\alpha < \beta$, which is representative of more predictable, but memory intensive real-time tasks, the fraction of time the real-time bridge can be opened decreases as the delay $D(s_i)$ becomes significant compared to $wcet_i - avg_i$. The performance of ADAPTIVE degrades more rapidly than BOUND, but it remains with a competitive ratio of 18%, which compares even more favorably than the MPEG case.

5.2 Predictable Execution Model

In this section, we detail the system model, architectural challenges (Section 5.2.1), programming constraints (Section 5.2.2), schedulability analysis (Section 5.2.3) and evaluation (Section 5.2.4) of the PRedictable Execution Model. The key issue in our interference analysis and on-line coscheduler is that when a typical real-time task is executed on a COTS CPU, cache misses are unpredictable, making it difficult to avoid low-level contention for access to main memory in the worst-case. To overcome this issue, PREM uses a set of compiler and OS techniques to predictably schedule all main memory accesses required by a task during specified time intervals.

We assume a single core system executing a set of N real-time periodic tasks $\Gamma = \{\tau_1, \ldots, \tau_N\}$. Each task can use one or more peripherals to transfer input or output data to or from main memory. We model all peripheral activities as a set of M periodic I/O flows $\Gamma^{\mathrm{I/O}} = \{\tau_1^{\mathrm{I/O}}, \ldots, \tau_M^{\mathrm{I/O}}\}$ with assigned timing reservations. The I/O management system in Section 2.1 is used to schedule I/O flows. As in Chapter 4, the code for each task τ_i is divided into a set of N_i scheduling intervals $\{s_{i,1}, \ldots, s_{i,N_i}\}$, which are executed sequentially at run-time. The timing requirements of τ_i can be expressed by a tuple $\{\{e_{i,1}, \ldots, e_{i,N_i}\}, p_i, D_i\}$, where p_i, D_i are the period and relative deadline of the task, with $D_i \leq p_i$, and $e_{i,j}$ is the maximum execution time of $s_{i,j}$, assuming that the interval runs in isolation with no memory interference. As usual, a job can only be preempted by a higher priority job at the end of a scheduling interval. This

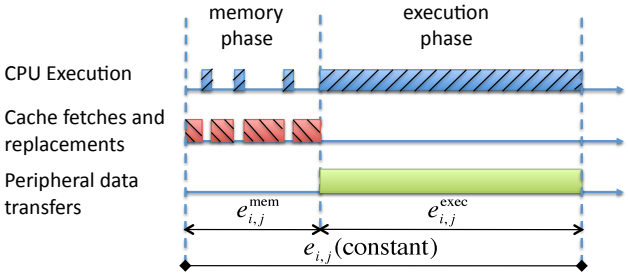


Figure 5.7: Predictable Interval with constant execution time.

ensures that the cache content can not be altered by the preempting job during the execution of an interval. We classify the scheduling intervals into *compatible intervals* and *predictable intervals*.

Compatible intervals are compiled and executed without any special provisions (they are backwards compatible). Cache misses can happen at any time during these intervals. The task code is allowed to perform OS system calls, but blocking calls must have bounded blocking time. Furthermore, the task can be preempted by interrupt handlers of associated peripherals. We assume that the maximum execution time $e_{i,j}$ for a compatible interval can be computed based on traditional static analysis techniques. However, to reduce the pessimism in the analysis, we prohibit peripheral traffic from being transmitted during a compatible interval. Ideally, there should be a small number of compatible intervals which are kept as short as possible.

Predictable intervals are specially compiled to execute according to the PREM model shown in Figure 5.7, and exhibit three main properties. First, each predictable interval is divided into two different phases. During the initial *memory phase*, the CPU accesses main memory to perform a set of cache line fetches and replacements. At the end of the memory phase, all cache lines required during the predictable interval are available in last level cache. Second, the second phase is know as the *execution phase*. During this phase, the task performs useful computation without suffering any last level cache misses. Predictable intervals do not contain any system calls and can not be preempted by interrupt handlers. Hence, the CPU does not perform any external main memory access during the execution phase. Due to this property, peripheral traffic can be scheduled during the execution phase of a predictable interval without causing any contention for access to main memory. Third, at run-time, we force the execution time of a predictable interval to be always equal to $e_{i,j}$. Let $e_{i,j}^{\text{mem}}$ be the maximum time required to complete the memory phase and $e_{i,j}^{\text{exec}}$ to complete the execution phase. Then offline we set $e_{i,j} = e_{i,j}^{\text{mem}} + e_{i,j}^{\text{exec}}$ and at run-time, even if the memory phase lasts for less than $e_{i,j}^{\text{mem}}$ time units, the overall interval still completes in exactly $e_{i,j}$. This property greatly increases task predictability without affecting CPU worst-case guarantees. In particular, as we show in Section 5.2.3, it ensures that hard real-time guarantees can be extended to I/O flows.

Figure 5.8 shows a concrete example of a system-level predictable schedule for a task set comprising two tasks τ_1, τ_2 together with two I/O flows $\tau_1^{\text{I/O}}, \tau_2^{\text{I/O}}$ which service τ_1 and τ_2 respectively. Both tasks and I/O flows are scheduled according to fixed priority, with τ_1 having higher priority than τ_2 and $\tau_1^{\text{I/O}}$ higher priority than $\tau_2^{\text{I/O}}$. We

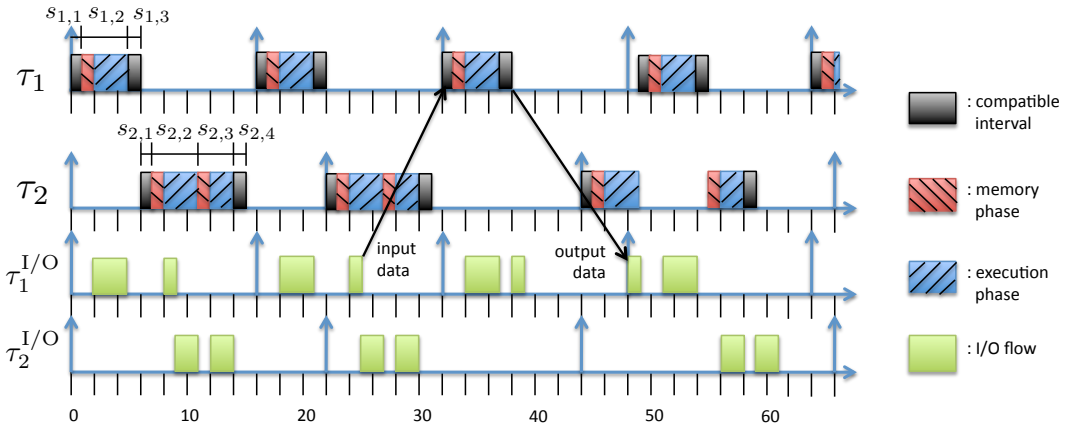


Figure 5.8: Example System-Level Predictable Schedule

set $D_i = p_i$ and assign to each I/O flow the same period and deadline as its serviced task and a transmission time equal to 4 time units. As shown in Figure 5.8 for task τ_1, this means that the input data for a given job is transmitted in the period before the job is executed, and the output data is transmitted in the period after. Task τ_1 has a single predictable interval of length $e_{1,2} = 4$ while τ_2 has two predictable intervals of lengths $e_{2,2} = 4$ and $e_{2,3} = 3$. The first and last interval of both τ_1 and τ_2 are special compatible intervals. These intervals are needed to execute the associated peripheral driver (including interrupt handlers) and set up the reception and transmission buffers in main memory (i.e. read and write system calls). More details are provided in Section 5.2.4. I/O flows can be scheduled both during execution phases and while the CPU is idle. As we will show in Section 5.2.3, the described scheme can be modeled as a hierarchical scheduling system [90], where the CPU schedule of predictable intervals supplies available transmission time to I/O flows. Therefore, existing tests can be reused to check the schedulability of I/O flows. However, due to the characteristics of predictable intervals, a more complex analysis is required to derive the supply function.

5.2.1 Architectural Constraints and Solutions

Predictable intervals are executed in a radically different way compared to the speculative execution model that COTS components are typically designed to support. In this section, we detail the challenges and solutions to implement the PREM execution model on top of a COTS architecture.

Caching and Prefetch

Our general strategy to implement the memory phase consists of two steps: (1) we determine the complete set of memory regions that are accessed during the interval. Each region is a continuous area in virtual memory. In general, its start address can only be determined at run-time, but its size A is known at compile time. (2) During the memory

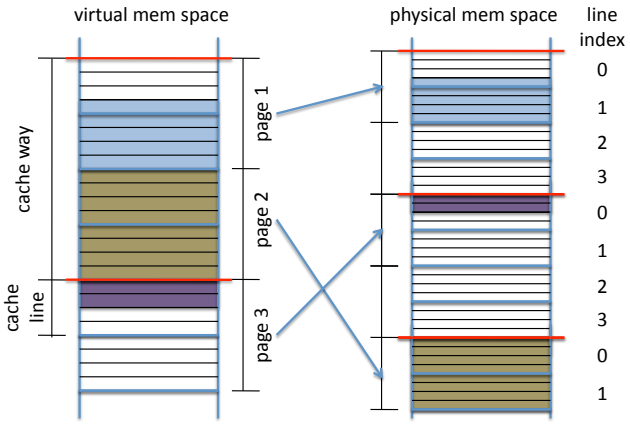


Figure 5.9: Cache organization with one memory region

phase, we prefetch all cache lines that contain instructions and data for required regions; most COTS instruction sets include a prefetch instruction that can be used to load specific cache lines in last level cache. Step (1) will be detailed in Section 5.2.2. Step (2) can be successful only if there is no cache *self-eviction*, that is, prefetching a cache line never evicts another line that has already been accessed (prefetched) during the same memory phase. In the remainder of this section, we describe self-eviction prevention.

Most COTS CPUs implement the last-level cache as an N-way set associative cache. Let B be the total size of the cache and L be the size of each cache line in bytes. Then the byte size of each of the N cache ways is $W = B/N$. An associative set is the set of all cache lines, one for each way, which have the same index in cache; there are W/L associative sets. Last level cache is typically physically tagged and physically indexed, meaning that cache lines are accessed based on physical memory addresses only. We also assume that last level cache is not exclusive, that is, when a cache line is copied to a higher cache level it is not removed from the last level. Figure 5.9 shows an example where $L = 4, W = 16$ (parameters are chosen to simplify the discussion and are not representative of typical systems). The main idea behind our conflict analysis is as follows: we compute the maximum amount of entries in each associative set that are required to hold the cache lines prefetched for all memory regions in a scheduling interval. Based on the cache replacement policy, in the following subsection we then derive a safe lower bound on the amount of entries that can be prefetched in an associative set without causing any self-eviction.

Consider a memory region of size A. The region will occupy at most $K = \lceil \frac{A-1}{L} \rceil + 1$ cache lines. As shown in Figure 5.9 for a region with $A = 15, K = 5$, the worst case is produced when the region uses a single byte in its first cache line. Assume now that virtual memory addresses coincide with physical addresses. Then since the region is contiguous and there are W/L cache lines in each way, the maximum number of entries used in any associative set by the region is $\lceil \frac{K}{W/L} \rceil$. For example, the region in Figure 5.9 requires two entries in the set with index 0. We then derive the maximum number of entries for the entire interval by summing the entries required for each memory region. Unfortunately, this is not generally true if the system employs paged virtual memory. If the page size P is

smaller than the size W of each way, the index of each cache line inside the cache way is different for virtual and physical addresses. In the example of Figure 5.9 with $P = 8$, the number of entries for the memory region is increased from 2 to 3. We consider two solutions: 1) if the system supports it, we can select a page size multiple of W just for our specific process. This solution, which we employed in our implementation, solves the problem because the index in cache for virtual and physical addresses is the same no matter the page allocation. 2) We use a modified page allocation algorithm in the OS. Cache-aware allocation algorithms have been proposed in the literature, for example in [57] for cache partitioning. Note that a suitable allocation algorithm could decrease the required number of associative entries by controlling the allocation in physical memory of multiple regions. We plan to pursue this solution in our future work.

Cache Replacement Analysis

We now consider the cache replacement policy. Let Q be the maximum number of entries in any associative set required by the predictable interval. Furthermore, let Q' be the number of such entries relative to cache lines that are accessed multiple times during the memory phase; in our implementation, this only includes the cache lines that contain the small amount of instructions of the memory phase itself, so $Q' = 1$. Based on the replacement policy and possibly Q', we now show how to compute a lower bound on Q such that no self-eviction can happen if Q is less than or equal to the derived bound.

We consider four different replacement policies: random, FIFO, LRU and pseudo-LRU, which cover most implemented cache architectures. Under *random* policy, whenever a new cache line is loaded in an associative set, the line to be evicted is chosen at random among all N lines in the set. While this policy is implemented in some modern COTS CPUs with very large cache associativity, such as Intel Core architecture, it is clearly ill suited to real-time systems. In *FIFO* policy, a FIFO queue is associated with each associative set as shown in Figure 5.10(a). Each newly fetched cache line is inserted at the head of the queue (position q_1), while the cache line which was previously at the back of the queue (position q_N) is evicted. In the figure, $l_1, \ldots, l_4$ represent cache lines used in the predictable interval, while dashes represent other cache lines available in cache but unrelated to the predictable interval. Finally, grayed boxes represent invalidated cache lines. The Least Recently Used (*LRU*) policy also uses a replacement queue, but a cache line is moved at the head of the queue whenever it is accessed. Due to the complexity of implementing LRU, an approximated version of the algorithm known as *pseudo-LRU* can be employed where N is a power of 2. A binary replacement tree with $N-1$ internal nodes and N leaves $q_1, \ldots, q_N$ is constructed as shown in Figure 5.11 for $N = 4$. Each internal node encodes a "right" or "left" direction using a single bit of data, while each leaf encodes a fetched cache line. Whenever a replacement decision must be taken, the encoded directions are followed starting from the root to the leaf representing the cache line to be replaced. Furthermore, whenever a cache line is accessed, all directions on the path from the root to the leaf of the accessed cache line are set to be the opposite of the followed path. Figure 5.11 shows an example where four cache lines are accessed in the order l_1, l_2, l_3, l_2, l_4, causing a self-eviction.

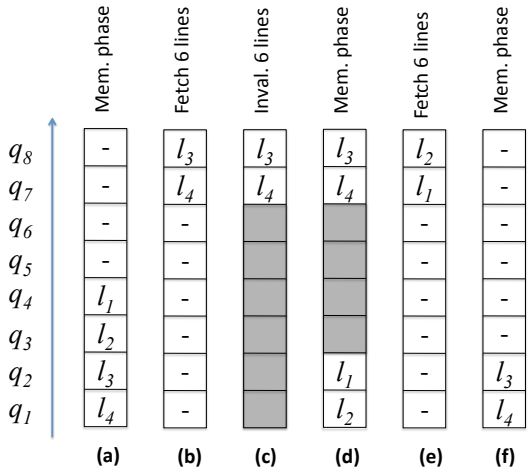


Figure 5.10: FIFO policy, example replacement queue.

Without loss of generality, in all proofs in this section we focus on the analysis of a single associative set with Q cache lines accessed during the predictable interval. Furthermore, let $l_1, \ldots, l_Q$ be the set of Q cache lines in the order in which they are first accessed during the memory phase (note that this order can be different from the order in which the lines are prefetched due to the Q' lines that are accessed multiple times). Results for LRU and random are simple and well-known, see [35] for example; we detail the bound derivation in the following theorem to provide a better understanding of the replacement policies.

Theorem 27 *A memory phase will not suffer any cache self-eviction in an associative set requiring Q entries, if Q is at most equal to:*

- 1*: for random replacement policy;*
- N*: for LRU replacement policy.*

Proof.

Clearly no self-eviction is possible if $Q = 1$, since after the unique cache line is fetched no other cache line can be accessed in the associative set.

Now consider LRU policy. When a cache line l_i is first accessed, it is fetched (if it is not already in cache) and moved to the beginning of the replacement queue. Since no cache line outside of $l_1, \ldots, l_Q$ is accessed in the associative set, when l_i is first accessed, cache lines $l_1, \ldots, l_{i-1}$ must be at the head of the queue in some order. But $Q \leq N$ implies $i - 1 \leq N - 1$, hence the algorithm always picks an unrelated line at the back of the queue to be replaced. □ Note that for random the bound of 1 is tight, since in the worst-case fetching a second cache line in the associative set can cause the first cache line to be evicted. Furthermore, note that if the cache is not invalidated between

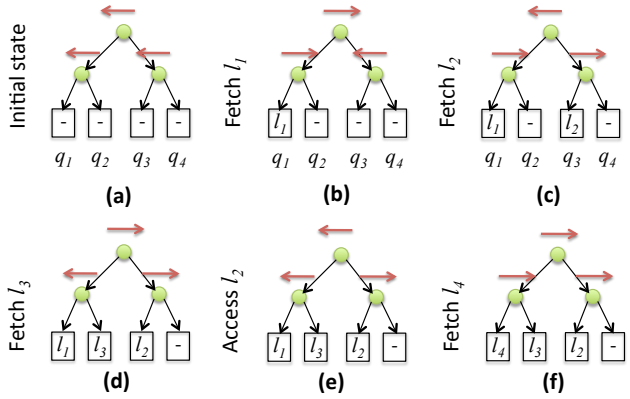


Figure 5.11: Pseudo-LRU policy, full-invalidation, $Q' = 1$.

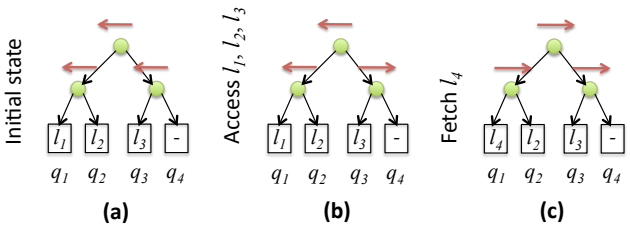


Figure 5.12: Pseudo-LRU policy, partial-invalidation.

successive activations of the same predictable interval, then some of the $l_1, \ldots, l_Q$ cache lines can still be available in cache. Assume that at the beginning of the predictable interval l_2 is available but not l_1. Then even with LRU policy, prefetching l_1 can potentially cause l_2 to be evicted. According to our definition, this is not a self-eviction: l_2 is evicted before it is accessed during the memory phase, and is then reloaded when it is first accessed, thus guaranteeing that all required lines are available in cache at the end of the memory phase. In fact, Theorem 27 proves that the bounds for random and LRU are independent of the state of the cache before the beginning of the predictable interval.

Unfortunately, the same is not true of FIFO and pseudo-LRU. Consider FIFO policy. If a cache line l_i is already available in cache, then when l_i is accessed during the memory phase no modification is performed on the replacement queue. In some situations this can cause l_i to be evicted by another line fetched after l_i in the memory phase, causing a self-eviction. Therefore, it is important to consider the state of the cache before the beginning of the predictable interval. To this end, we consider two different *cache invalidation models*. In the *full-invalidation* model, none of the cache lines used by a predictable interval are already available in cache at the beginning of the memory phase. Furthermore, the replacement policy is not affected by the presence of invalidated cache lines. As an example, this model can be enforced by fully invalidating the cache either at the end or at the beginning of each predictable interval, which in most architectures has the side effect of resetting all replacement queues/trees. While this model is more predictable, several COTS CPUs do not support it, including our testbed described in Section 5.2.4. In the *partial-invalidation* model, selected lines in last level cache can be invalidated, for example to handle data coherency between

the CPU and peripherals in compatible intervals. Furthermore, the replacement policy always selects invalidated cache lines before valid lines, independently of the state of the queue/tree.

A detailed analysis of FIFO replacement is provided in [40]; here we summarize the results and intuitions relevant to our system.

Theorem 28 (Directly follows from Lemma 1 in [40]) *Assume FIFO replacement with full-invalidation semantic. Then no self-eviction is possible for an associative set with $Q \leq N$.*

The analysis for partial-invalidation FIFO is more complex. Figure 5.10 (analogous to Figure 2 in [40]) depicts a clarifying example, where the "mem. phase", "fetch" and "inval." labels show the state of the replacement queue after the memory phase and after some cache lines have been either fetched or invalidated outside the predictable interval, respectively. Note that after Step (e), lines l_1, l_2 remain in cache but not l_3, l_4. Hence, when l_3 and l_4 are fetched during the next memory phase in Step (f), they evict l_1 and l_2. The solution is to prefetch $l_1, \ldots, l_Q$ multiple times in the memory phase, each time in the same order.

Theorem 29 (Directly follows from Theorem 3 in [40]) *Assume FIFO replacement with partial-invalidation semantic. If the Q cache lines in an associative set, with $Q \leq N$, are prefetched at least Q times in the same access-order $l_1, \ldots, l_Q$, then all Q cache lines are available in cache at the end of the memory phase. Furthermore, no more than Q fetches are required.*

Note that while executing the prefetching code Q times might seem onerous, in practice the majority of the time in the memory phase is spent fetching and writing back cache lines, and Theorem 29 ensures that no more than Q fetches and write backs are required for each associative set.

While LRU and FIFO allow to prefetch up to N cache lines with N fetches, the same is not true of pseudo-LRU, even in the full-invalidation model. Consider again Figure 5.11, where no relevant lines are available in cache at the beginning of the memory phase but $Q' = 1$, meaning that line l_2 can be accessed multiple times. Then it is easy to see that no more than 3 cache lines can be prefetched without causing a self-eviction. The key intuition is that up to Step (d), and with respect to l_1, l_2, l_3, the replacement tree encodes the LRU order exactly. However, after l_2 is accessed again at Step (e), the LRU order is lost, and the next replacement causes l_1 to be evicted instead of the remaining unrelated cache line. Furthermore, note that the strategy employed in Theorem 29 does not work in this case, because l_1 has already been fetched during the memory interval before being evicted. A similar situation can happen even if $Q' = 0$ in the partial-invalidation model, as shown in Figure 5.12. Assume that due to partial replacements and cache accesses, at the beginning of the memory phase the cache state is as shown in Figure 5.12(a), with l_1, l_2 and l_3 being available in cache but not l_4. Then after the first three cache lines are prefetched in order, l_1 will be replaced when l_4 is next prefetched.

A lower bound that is independent of Q' and of the invalidation model was first proven in [81].

Theorem 30 (Theorem 10 in [81]) *Under pseudo-LRU replacement, no self-eviction is possible for an associative set with* $Q \leq \log_2 N + 1$.

We now show in Theorem 33 that under the full-invalidation model, a better bound can be obtained based on the value of Q'. In the theorem, a subtree of a pseudo-LRU replacement tree is a tree rooted at any internal node of the replacement tree. For example, the replacement tree in Figures 5.11, 5.12 has three subtrees: the whole tree, which has height 2 and leaves $q_1, \ldots, q_4$, and its left and right subtrees with height 1 and leaves q_1, q_2 and q_3, q_4, respectively. For each subtree of height k, we compute a lower bound Lb_k^p on the number of cache lines in $l_1, \ldots, l_Q$ that can be allocated in the subtree (either because they are already available in cache at the beginning of the predictable interval or because they are fetched during the memory phase) without causing any self-eviction, assuming that none of the lines were available in cache at the beginning of the predictable interval and that up to p lines can be accessed multiple times during the memory phase. Note that by definition, $Lb_k^i \leq Lb_k^j$ if $i \geq j$. Furthermore, $\forall p, Lb_1^p = 2$ by Theorem 30. The following Lemmas 31, 32 prove some important facts on Lb_k^p.

Lemma 31 $Lb_k^0 = 2Lb_{k-1}^0$.

Proof.

Consider once again the left and right subtrees with height $k-1$. By contradiction and without loss of generality, assume that a self-eviction happens in the left subtree; then at least $Lb_{k-1}^0 + 1$ cache lines must be allocated in it. Since no relevant cache line is originally available in cache and furthermore no cache line is accessed more than once, it follows that every access results in a fetch and replacement. Furthermore, whenever a cache line is fetched in the left subtree, the root of the height-k subtree is changed to point to the right subtree and viceversa. Hence, to fetch $Lb_{k-1}^0 + 1$ lines in the left subtree, at least Lb_{k-1}^0 lines must be fetched in the right subtree. Therefore, at least $2Lb_{k-1}^0 + 1$ total cache lines must be allocated in the whole height-k subtree to cause a self-eviction, implying that $Lb_k = 2Lb_{k-1}^0$ is a valid lower bound. □

Note that from Lemma 31 and $Lb_1^0 = 2$ it immediately follows that $Lb_k^0 = 2^k$.

Lemma 32 *For any subtree of height* $k > 1$ *with* $p > 0$, $Lb_k^p = \min(Lb_{k-1}^{p-1} + 1, 2Lb_{k-1}^p)$.

Proof.

Consider the left and right subtrees with height $k-1$, and assume that out of the p cache lines that can be accessed multiple times, i are allocated in the left subtree and j are allocated in the right subtree, with $i+j = p$. By contradiction and without loss of generality, assume that a self-eviction happens in the left subtree; then at least $Lb_{k-1}^i + 1$ cache lines must be allocated in it. We distinguish two cases.

Case (1): $j > 0$. Then following the same reasoning as in Theorem 30, it is sufficient to allocate a single cache line in the right subtree, resulting in a total of $Lb^i_{k-1} + 2$ lines required to cause a self eviction; note that this value is minimized by maximizing i, e.g. when $j = 1, i = p - 1$, resulting in $Lb^{p-1}_{k-1} + 2$ required cache lines.

Case (2): $j = 0$. Then following the same reasoning as in Lemma 31, at least Lb^i_{k-1} cache lines must be allocated in the right subtree, for a total of $2Lb^i_{k-1} + 1 = 2Lb^p_{k-1} + 1$ cache lines.

Combining case (1) and (2), it follows that no eviction is possible if the number of allocated lines is at most equal to $\min(Lb^{p-1}_{k-1} + 1, 2Lb^p_{k-1})$, which concludes the proof. □

We can finally prove Theorem 33.

Theorem 33 *Under pseudo-LRU replacement with full-invalidation semantic, no self-eviction is possible if:*

$$Q \leq \begin{cases} \frac{N}{2^{Q'}} + Q' & \text{if } Q' < \log_2 N; \\ \log_2 N + 1 & \text{otherwise.} \end{cases}$$

Proof.

The proof proceeds by induction on the pair (p, k) ordered by p first, e.g. in the sequence $(0, 1), \ldots, (0, Q), (1, 1), \ldots, (1, Q), \ldots, (Q', 1), \ldots, (Q', Q)$. In particular, we show that the following property holds $\forall p, k$:

$$Lb^p_k = \begin{cases} 2^{k-p} + p & \text{if } p < k; \\ k + 1 & \text{otherwise.} \end{cases}$$

Since a pseudo-LRU replacement tree for a N-associative set has height $\log_2 N$, the theorem then follows.

By Lemma 31, the property is verified for $p = 0$ since $Lb^0_k = 2^k$. By Theorem 30, the property is verified for $k = 1$ since $Lb^p_1 = 2$. Therefore, it remains to complete the induction step by showing that the property holds at step (p, k) with $p > 0, k > 1$. We do so by applying Lemma 32, assuming that the property also holds at previous steps $(p - 1, k - 1)$ and $(p, k - 1)$. We consider three cases based on the relative value of p, k.

Case (1): $p < k - 1$. We have to show that $Lb^p_k = 2^{k-p} + p$. Then:

$$Lb^p_k = \min\left(2^{(k-1)-(p-1)} + (p-1) + 1, 2(2^{(k-1)-p} + p)\right) = \min\left(2^{k-p} + p, 2^{k-p} + 2p\right) = 2^{k-p} + p.$$

Case (2): $p = k - 1$. We have to show that $Lb^p_k = 2^{k-p} + p = k + 1$. Then:

$$Lb^p_k = \min\left(2^{(k-1)-(p-1)} + (p-1) + 1, 2((k-1)+1)\right) = \min\left(2^{k-p} + p, 2k\right) = k + 1.$$

Case (3): $p \geq k$. We have to show that $Lb^p_k = k + 1$. Then:

$$Lb^p_k = \min\left((k-1) + 1 + 1, 2((k-1)+1)\right) = \min\left(k + 1, 2k\right) = k + 1.$$

□

Computing Phase Length

To provide predictable timing guarantees, the maximum time required for memory phase $e_{i,j}^{\text{mem}}$ and for execution phase $e_{i,j}^{\text{exec}}$ must be computed. We assume that an upper bound to $e_{i,j}^{\text{exec}}$ can be derived based on static analysis of the execution phase. Note that our model only ensures that all required instructions and data are available in last level cache; cache misses can still occur in higher cache levels. Analyses that derive patterns of cache misses for split level 1 data and instruction caches have been proposed in [80, 64]. These analyses can be safely executed because according to our model, level 1 misses during the execution phase will not require the CPU to perform any external access. $e_{i,j}^{\text{mem}}$ depends on the time required to prefetch all accessed memory regions. Note that since the task can be preempted at the boundary of scheduling intervals, the cache state is unknown at the beginning of the memory phase. Hence, each prefetch could require both a cache line fetch and a replacement in last cache level. An analysis to compute upper bounds for read/write operations using a COTS DRAM memory controller is detailed in [68]. Note that since the number and relative addresses of prefetched cache lines is known, the analysis could potentially exploit features such as burst read and parallel bank access that are common in modern memory controllers.

Finally, for systems implementing paged virtual memory, we employ the following three assumptions: 1) the CPU supports hardware Translation Lookaside Buffer (TLB) management; 2) all pages used by predictable intervals are locked in main memory; 3) the TLB is large enough to contain all page entries for a predictable interval without suffering any conflict. Under such assumptions, each page used in a predictable interval can cause at most one TLB miss during the memory phase, which requires a number of fetches in main memory equal to at most the level of the page table.

Interval Length Enforcement

As described in Section 5.2, each predictable interval is required to execute for exactly $e_{i,j}$ time units. If $e_{i,j}$ is set to be at least $e_{i,j}^{\text{mem}} + e_{i,j}^{\text{exec}}$, the computation in the execution phase is guaranteed to terminate at or before the end of the interval. The interval can then enter active wait until $e_{i,j}$ time units have elapsed since the beginning of its memory phase. In our implementation, elapsed time is computed using a CPU performance counter that is directly accessible by the task; therefore, neither OS nor memory interaction are required to enforce interval length.

Scheduling synchronization

In our model, peripherals are only allowed to transmit during a predictable interval's execution phase or while the CPU is idle. To compute the peripheral schedule, the peripheral scheduler must thus know the status of the CPU schedule. Synchronization can be achieved by connecting the peripheral scheduler to a peripheral interconnection. Scheduling messages can then be sent by either a task or the OS to the peripheral scheduler. In particular, at the end of each memory phase the task sends to the peripheral scheduler the remaining amount of time until the end of the current predictable interval. Note that propagating a message through the interconnection takes non-zero time. Since this time

is typically negligible compared to the size of a scheduling interval[1], we will ignore it in the rest of our discussion. However, the schedulability analysis of Section 5.2.3 could be easily modified to take such overhead into account.

Finally, to avoid executing interrupt handlers during predictable intervals, a peripheral should only raise interrupts to the CPU during compatible intervals of its serviced task. As we describe in Section 5.2.4, in our I/O management scheme peripherals raise interrupts through their assigned real-time bridge. Since the peripheral scheduler communicates with each real-time bridge, it is used to block interrupt propagation outside the desired compatible intervals. Note that blocking real-time bridge interrupts to the CPU will not cause any loss of input data because the real-time bridge is capable of independently acknowledging the peripheral and storing all incoming data in the bridge local buffer.

5.2.2 Programming Model

Our system supports COTS applications written in standard high-level languages such as C. Unmodified code can be executed within one or more compatible intervals. To create predictable intervals, programmers add source code annotations as C preprocessor macros. The PREM real-time compiler creates code for predictable intervals so that it does not incur cache misses during the execution phase and the interval itself has a constant execution time.

Due to current limitations of our static code analysis, we assume that the programmer manually partitions the task into intervals. In particular, compatible intervals can be handled in the conventional way while each predictable interval is encapsulated into a single function. All code within the function, and any functions transitively called by this function is executed as a single predictable interval. To correctly prefetch all code and data used by a predictable interval, we impose several constraints upon its code:

1. Only scalar and array-based memory accesses should occur within a predictable interval; there should be no use of pointer-based data structures.

2. The code can use data structures, in particular arrays, that are not local to functions in the predictable intervals, e.g. they are allocated either in global memory or in the heap[2]. However, the programmer should specify the first and last address that is accessed within the predictable interval for each non-local data structure. In general, it is difficult for the compiler to determine the first and last access to an array within a piece of code. The compiler needs this information to be able to load the relevant portion of each array into the last-level cache.

3. The functions within a predictable interval should not be called recursively. As described below, the compiler will inline callees into callers to make all the code within an interval contiguous in virtual memory. Furthermore, no system calls should be made by code within a predictable interval.

[1] In our implementation we measured an upper bound to the message propagation time of 1us, while we envision scheduling intervals with a length of 100-1000us.

[2] Note that data structures in the heap must have been previously allocated during a compatible interval.

4. Code within a predictable interval may only have direct calls. This alleviates the needs for pointer-analysis to determine the targets of indirect function calls; such analysis is usually imprecise and would bloat the code within the interval.

5. No stack allocations should occur within loops. Since all variables must be loaded into the cache at function entry, it must be possible for the compiler to safely hoist the allocations to the beginning of the function which initiates the predictable interval.

While these constraints may seem restrictive, some of these features are rarely used in real-time C code e.g., indirect function calls, and the others are met by many types of functions. We believe that the benefit of faster, more predictable behavior for program hot-spots outweighs the restrictions imposed by our programming model. Furthermore, existing code that is too complex to be compiled into predictable intervals can still be executed inside compatible intervals. Therefore, our model permits a smooth transition for legacy systems.

Notice that, the compiler can be used to verify that all of the aforementioned restrictions are met. Simple static analysis can determine whether there is any irregular data structure usage, indirect function calls, or system calls. During compilation, the compiler employs several transforms to ensure that code marked as being within a predictable interval does not cause a cache miss. First, the compiler inlines all functions called (either directly or transitively) during the interval into the top-level function defining the interval. This ensures that all program analysis is intra-procedural and that all the code for the interval is contiguous within virtual memory. Second, the compiler can transform the program so that all cache misses occur during the memory phase, which is located at the beginning of the predictable scheduling interval. To be specific, it inserts code after the function prologue to prefetch the code and data needed to execute the interval. Based on the described constraints, this includes three types of contiguous memory regions: (1) the code for the function; (2) the actual parameters passed to the function and the stack frame (which contains local variables and register spill slots); and (3) the data structures marked by the programmer as being accessed by the interval. Third, the compiler inserts code to send scheduling messages to the peripheral scheduler as will be described in Section 5.2.4. Fourth, the compiler emits code at the end of the predictable interval to enforce its constant length. In particular, the compiler identifies all return instructions within the function and adds the required code before them. Finally, based on the information on prefetched memory regions, we assume that an external tool, such as a static timing analyzer used to compute maximum phase length, can check the absence of cache self-evictions according to the analysis provided in Section 5.2.1.

5.2.3 Schedulability Analysis

PREM allows us to enforce strict timing guarantees for both CPU tasks and their associated I/O flows. By setting timing parameters as shown in Figure 5.8, the task schedule becomes independent of I/O scheduling. Therefore, task schedulability can be checked using available schedulability tests. As an example, assume that tasks are scheduled

according to fixed priority scheduling as in Figure 5.8. For a task τ_i, let $e_i = \sum_{j=1}^{N_i} e_{i,j}$ be the sum of the execution times of its scheduling intervals, or equivalently, the execution time of the whole task. Furthermore, let $\mathrm{hp}_i \subset \Gamma$ be the set of higher priority tasks than τ_i, and lp_i the set of lower priority tasks. Since scheduling intervals are executed non preemptively, τ_i can suffer a blocking time due to lower priority tasks of at most $B_i = \max_{\tau_l \in \mathrm{lp}_i} \max_{j=1\ldots N_l} e_{l,j}$. The worst-case response time of τ_i can then be found [19] as the fixed point r_i of the iteration:

$$r_i^{k+1} = e_i + B_i + \sum_{l \in \mathrm{hp}_i} \left\lceil \frac{r_i^k}{p_l} \right\rceil e_l, \tag{5.4}$$

starting from $r_i^0 = e_i + B_i$. Task set Γ is schedulable if $\forall \tau_i : r_i \leq D_i$.

We now turn our attention to peripheral scheduling. Assume that each I/O flow $\tau_i^{\mathrm{I/O}}$ is characterized by a maximum transmission time $e_i^{\mathrm{I/O}}$ (with no interference in both main memory and the interconnect), period $p_i^{\mathrm{I/O}}$ and relative deadline $D_i^{\mathrm{I/O}}$, where $D_i^{\mathrm{I/O}} \leq p_i^{\mathrm{I/O}}$. The schedulability analysis for I/O flows is more complex because the scheduling of data transfers depends on the task schedule. To solve this issue, we extend the hierarchical scheduling framework proposed by Shin and Lee in [90]. In this framework, tasks/flows in a *child scheduling model* execute using a timing resource provided by a *parent scheduling model*. Schedulability for the child model can be tested based on the *supply bound function* $\mathrm{sbf}(t)$, which represents the minimum resource supply provided to the child model in any interval of time t. In our model, the I/O flow schedule is the child model and $\mathrm{sbf}(t)$ represents the minimum amount of time in any interval of time t during which the execution phase of a predictable interval is scheduled or the CPU is idle. Define the *service time bound function* $\mathrm{tbf}(t)$ as the pseudo-inverse of $\mathrm{sbf}(t)$, that is, $\mathrm{tbf}(t) = \min\{x | \mathrm{sbf}(x) \geq t\}$. Then if I/O flows are scheduled according to fixed priority, in [90] it is shown that the response time $r_i^{\mathrm{I/O}}$ of flow $\tau_i^{\mathrm{I/O}}$ can be computed according to the iteration:

$$r_i^{\mathrm{I/O},k+1} = \mathrm{tbf}\Big(e_i^{\mathrm{I/O}} + \sum_{l \in \mathrm{hp}_i^{\mathrm{I/O}}} \left\lceil \frac{r_i^{\mathrm{I/O},k}}{p_l^{\mathrm{I/O}}} \right\rceil e_l^{\mathrm{I/O}}\Big), \tag{5.5}$$

where $\mathrm{hp}_i^{\mathrm{I/O}}$ has the same meaning as hp_i. In the remainder of this section, we detail how to compute $\mathrm{sbf}(t)$.

For the sake of simplicity, let us initially assume that tasks are strictly periodic and that the initial activation time of each task is known. Furthermore, notice that using the solution described in Section 5.2.1, we could enforce interval lengths not just for predictable intervals, but also for all compatible intervals. Finally, let h be the hyperperiod of task set Γ, defined as the least common multiple of all tasks' periods. Under these assumptions, it is easy to see that if Γ is feasible, the CPU schedule can be computed offline and repeats itself identically with period h after an initial interval of $2h$ time units (h time units if all tasks are activated simultenously). Therefore, a tight $\mathrm{sbf}(t)$ can be computed as the minimum amount of supply (time during which the CPU is idle or in execution phase of a predictable interval) during any interval of time t in the periodic task schedule, starting from the initial two hyperperiods. More formally, let $\{t^1, \ldots, t^K\}$ be the set of start times for all scheduling intervals activated in the first two hyperperiods; the following theorem shows how to compute $\mathrm{sbf}(t)$.

Theorem 34 *Let* $\mathrm{sf}(t', t'')$ *be the amount of supply provided in the periodic task schedule during interval* $[t', t'']$. *Then:*

$$\mathrm{sbf}(t) = \min_{k=1\ldots K} \mathrm{sf}(t^k, t^k + t). \tag{5.6}$$

Proof.

Let t', t'' be two consecutive interval start times in the periodic schedule. We show that $\forall \bar{t}, t' \leq \bar{t} \leq t'' : \min\big(\mathrm{sf}(t', t'+t), \mathrm{sf}(t'', t''+t)\big) \leq \mathrm{sf}(\bar{t}, \bar{t}+t)$. This implies that to minimize $\mathrm{sbf}(t)$, it suffices to check the start times of all scheduling intervals.

Assume first that $\bar{t}$ falls inside a compatible interval or a memory phase. Then the schedule provides no supply in $[t', \bar{t}]$. Therefore: $\mathrm{sf}(t', t' + t) = \mathrm{sf}(\bar{t}, t' + t) \leq \mathrm{sf}(\bar{t}, \bar{t} + t)$. Now assume that $\bar{t}$ falls in an execution phase or in an idle interval before the start time t'' of the next scheduling interval. Then $\mathrm{sf}(\bar{t}, t'') = t'' - \bar{t}$. Therefore: $\mathrm{sf}(t'', t'' + t) \leq \mathrm{sf}(t'', t'' + t - (t'' - \bar{t})) + (t'' - \bar{t}) = \mathrm{sf}(t'', \bar{t} + t) + \mathrm{sf}(\bar{t}, t'') = \mathrm{sf}(\bar{t}, \bar{t} + t)$.

To conclude the proof, it suffices to note that since the schedule is periodic, if $t' \geq 2h$, then $\mathrm{sf}(t', t' + t) = \mathrm{sf}(t^k, t^k + t)$ where $t^k = (t' \mod h) + h$. □

Unfortunately, the proposed $\mathrm{sbf}(t)$ derivation can only be applied to strictly periodic tasks. If Γ includes any sporadic task τ_i with minimum interarrival time p_i, the schedule can not be computed offline. Therefore, we now propose an alternative analysis that is independent of the CPU scheduling algorithm and computes a lower bound $\mathrm{sbf}_L(t)$ to $\mathrm{sbf}(t)$. Note that since $\mathrm{sbf}(t)$ is the minimum supply in any interval t, using Equation 5.5 with $\mathrm{sbf}_L(t)$ instead of $\mathrm{sbf}(t)$ will still result in a sufficient schedulability test.

Let $\overline{\mathrm{sbf}}(t)$ be the maximum amount of time in which the CPU is executing either a compatible interval or the memory phase of a predictable interval in any time window of length t. Then by definition, $\overline{\mathrm{sbf}}(t) = t - \mathrm{sbf}(t)$. Our analysis derives an upper bound $\overline{\mathrm{sbf}}_U(t)$ to $\overline{\mathrm{sbf}}(t)$. Therefore, $\mathrm{sbf}_L(t) = t - \overline{\mathrm{sbf}}_U(t)$ is a valid lower bound for $\mathrm{sbf}(t)$. For a given interval size t, we compute $\overline{\mathrm{sbf}}(t)$ using the following key idea. For each task τ_i, we determine the minimum and maximum amount of time $E_i^{\min}(t), E_i^{\max}(t)$ that the task can execute in a time window of length t while meeting its deadline constraint. Let $t_i, E_i^{\min}(t) \leq t_i \leq E_i^{\max}(t)$, be the amount of time that τ_i actually executes in the time window. Since the CPU is a single shared resource, it must hold $\sum_{i=1}^{N} t_i \leq t$ independently of the task scheduling algorithm. Finally, for each task τ_i, we define the *memory bound function* $\mathrm{mbf}_i(t_i)$ to be the maximum amount of time that the task can spend in compatible intervals and memory phases, assuming that it executes for t_i time units in the time window. We can then obtain $\overline{\mathrm{sbf}}_U(t)$ as the maximum over all feasible $\{t_1, \ldots, t_N\}$ tuples of $\sum_{i=1}^{N} \mathrm{mbf}_i(t_i)$. In other words, we can compute $\overline{\mathrm{sbf}}_U(t)$ by solving the following optimization problem:

$$\overline{\mathrm{sbf}}_U(t) = \max \sum_{i=1}^{N} \mathrm{mbf}_i(t_i), \tag{5.7}$$

$$\sum_{i=1}^{N} t_i \leq t, \tag{5.8}$$

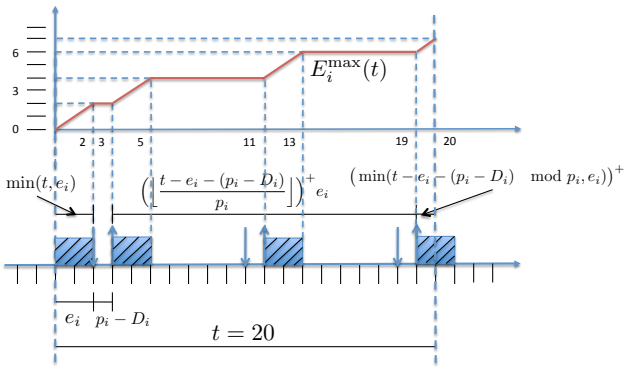


Figure 5.13: Derivation of $E_i^{\max}(t)$, periodic/sporadic task.

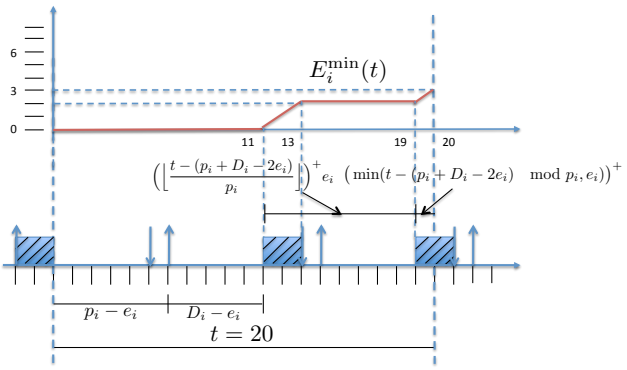


Figure 5.14: Derivation of $E_i^{\min}(t)$, periodic task.

$$\forall i, 1 \leq i \leq N : E_i^{\min}(t) \leq t_i \leq E_i^{\max}(t). \tag{5.9}$$

$E_i^{\max}(t)$ and $E_i^{\min}(t)$ can be computed according to the scenarios shown in Figures 5.13, 5.14 where the notation $(x)^+$ is used for $\max(x, 0)$.

Theorem 35 *For a periodic or sporadic task:*

$$E_i^{\max}(t) = \min(t, e_i) + \left(\left\lfloor \frac{t - e_i - (p_i - D_i)}{p_i} \right\rfloor\right)^+ e_i + \left(\min(t - e_i - (p_i - D_i) \bmod p_i, e_i)\right)^+. \tag{5.10}$$

For a sporadic task, $E_i^{\min}(t) = 0$, while for a periodic task:

$$E_i^{\min}(t) = \left(\left\lfloor \frac{t - (p_i + D_i - 2e_i)}{p_i} \right\rfloor\right)^+ e_i + \left(\min(t - (p_i + D_i - 2e_i) \bmod p_i, e_i)\right)^+.$$

Proof.

As shown in Figure 5.13, the relative distance between successive executions of a task τ_i is minimized when the task finishes at its deadline in the first period within the time window of length t, and as soon as possible (e.g., e_i time units after its activation) for each successive period. Equation 5.10 directly follows from this activation pattern, assuming that the time window coincides with the start time of the first job of τ_i. In particular:

- term $\min(t, e_i)$ represents the execution time for the first job, which is activated at the beginning of the time window;
- term $\left(\left\lfloor \frac{t-e_i-(p_i-D_i)}{p_i} \right\rfloor\right)^+ e_i$ represents the execution time of jobs in periods that are fully contained within the time window; note that the first period starts $e_i + (p_i - D_i)$ time units after the beginning of the time window;
- finally, term $\left(\min(t - e_i - (p_i - D_i) \bmod p_i, e_i)\right)^+$ represents the execution time for the last job in the time window.

Similarly, as shown in Figure 5.14, the relative distance between successive executions of a periodic job τ_i is maximized when the task finishes as soon as possible in its first period and as late as possible in all successive periods.

Equation 5.11 follows assuming that the time window coincides with the finishing time of the first job of τ_i, noticing that periodic task activations start $(p_i - e_i) + (D_i - e_i) = p_i + D_i - 2e_i$ time units after the beginning of the time window. Finally, $E_i^{\min}(t)$ is zero for a sporadic task since by definition the task has no maximum interarrival time. □

Lemma 36 *The optimization problem of Equations 5.7-5.9 admits solution if task set* Γ *is feasible.*

Proof.

The optimization problem admits solution if the set of t_i values that satisfy Equations 5.8 and 5.9 is not empty. Note that by definition $E_i^{\max}(t) \geq E_i^{\min}(t)$, therefore there is at least one admissible solution iff $\sum_{i=1}^{N} E_i^{\min}(t) \leq t$. Define $E_{Ui}^{\min}(t) = \big(t - (D_i - e_i)\big)^+ \frac{e_i}{p_i}$. Then it is easy to see that $\forall t, E_i^{\min}(t) \leq E_{Ui}^{\min}(t)$: in particular, both functions are equal to e_i for $t = p_i + D_i - e_i$ and increase by e_i every p_i afterwards. Now note that $E_{Ui}^{\min}(t) \leq t\frac{e_i}{p_i}$, therefore it also holds: $\sum_{i=1}^{N} E_i^{\min}(t) \leq t\sum_{i=1}^{N} \frac{e_i}{p_i}$. The proof follows by noticing that since Γ is feasible, it must hold $\sum_{i=1}^{N} \frac{e_i}{p_i} \leq 1$. □

$\mathrm{mbf}_i(t_i)$ can be computed using a strategy similar to the one in Theorem 34. Since t_i represents the amount of time that τ_i executes, we consider a schedule in which τ_i is executed continuously being activated every e_i time units, as shown in Figure 5.15. The start time of the first scheduling interval in the first job of τ_i is $t^1 = 0$, the start time of the second scheduling interval is $t^2 = e_{i,1}$, and so on and so forth until the first interval of the second job which has start time equal to e_i. Then $\mathrm{mbf}_i(t_i)$ can be computed as the maximum amount of time that the task spends in a compatible interval or memory phase in any time window t_i.

Theorem 37 *Let* $\mathrm{mf}_i(t', t'')$ *be the amount of time that task* τ_i *spends in a compatible interval or memory phase in the interval* $[t', t'']$*, in the schedule in which* τ_i *is executed continuously. Then:*

$$\mathrm{mbf}_i(t_i) = \max_{k=1\ldots N_i} \mathrm{mf}_i(t^k, t^k + t_i). \tag{5.11}$$

Proof.

Note that the schedule in which τ_i executes continuously is periodic with period p_i. The same strategy as in Theorem 34 can be used to show that if t', t'' are consecutive interval start times, then $\forall \bar{t}, t' \leq \bar{t} \leq t'' : \max\big(\mathrm{mf}_i(t', t' + t_i), \mathrm{mf}_i(t'', t'' + t_i)\big) \geq \mathrm{mf}_i(\bar{t}, \bar{t} + t_i)$. □

As an example, in Figure 5.15 $\mathrm{mbf}_i(t_i)$ is maximized in the time window that starts at $t^4 = 10$.

Since $\mathrm{mbf}_i(t_i)$ is a nonlinear function, computing $\overline{\mathrm{sbf}}_U(t)$ according to Equations 5.7-5.9 requires solving a non-linear optimization problem. To simplify the problem, we consider a linear upper bound approximation $\mathrm{mbf}_{Ui}(t_i) = \alpha_i t_i + \delta_i$, as shown in Figure 5.15, where

$$\alpha_i = \Big(\sum_{\forall s_{i,j}\ \text{compatible}} e_{i,j} + \sum_{\forall s_{i,j}\ \text{predictable}} e_{i,j}^{\text{mem}}\Big)/e_i, \tag{5.12}$$

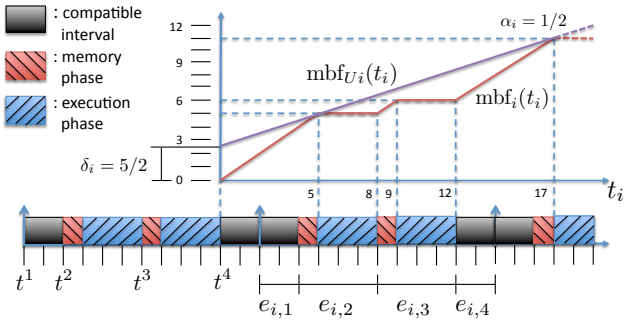


Figure 5.15: Derivation of $\mathrm{mbf}_i(t_i)$ and $\mathrm{mbf}_{Ui}(t_i)$.

and δ_i is the minimum value such that $\forall t_i, \mathrm{mbf}_{Ui}(t_i) \geq \mathrm{mbf}_i(t_i)$. Using $\mathrm{mbf}_{Ui}(t_i) = \alpha_i t_i + \delta_i$, the optimization problem of Equations 5.7-5.9 can be rewritten as follows:

$$\overline{\mathrm{sbf}}_U(t) = \sum_{i=1}^{N}(\delta_i + \alpha_i E_i^{\min}(t)) + \max \sum_{i=1}^{N} \alpha_i x_i, \tag{5.13}$$

$$\sum_{i=1}^{N} x_i \leq t - \sum_{i=1}^{N} E_i^{\min}(t), \tag{5.14}$$

$$\forall i, 1 \leq i \leq N : 0 \leq x_i \leq E_i^{\max}(t) - E_i^{\min}(t), \tag{5.15}$$

where we substituted $x_i = t_i - E_i^{\min}(t)$. Without loss of generality, assume that the N tasks are ordered by non-increasing values of α_i. Then Algorithm 8 computes the solution val to Equations 5.7-5.9. The algorithm first computes the time bound $t - \sum_{i=1}^{N} E_i^{\min}(t)$ on the sum of the variables x_i. Then, starting from x_1, it assigns to each variable the minimum between the remaining time t_{rem} and the upper constraint of Equation 5.15. Since the tasks are ordered by non-increasing values of α_i, it is trivial to see that Algorithm 8 computes the maximum of Equation 5.13. Furthermore, the while loop at Lines 5-10 is executed at most N times, and each iteration requires constant time. Hence, the algorithm has complexity $O(N)$.

Algorithm 8 computes $\overline{\mathrm{sbf}}_U(t)$ for a specific value of t. To test the schedulability condition of Equation 5.5 using the upper bound $tbf_U(t) = \min\{x | \mathrm{sbf}_L(x) \geq t\}$, $\overline{\mathrm{sbf}}_U(t)$ must be computed for all t in the interval $[0, \max_i D_i^{\mathrm{I/O}}]$. The reason is as follows. Note that if for any i:

$$\mathrm{tbf}_U\left(e_i^{\mathrm{I/O}} + \sum_{l \in \mathrm{hp}_i^{\mathrm{I/O}}} \left\lceil \frac{r_i^{\mathrm{I/O},k}}{p_l^{\mathrm{I/O}}} \right\rceil e_l^{\mathrm{I/O}}\right) > \max_i D_i^{\mathrm{I/O}}, \tag{5.16}$$

then the system is not schedulable. By definition of $\mathrm{tbf}_U(t)$, if $e_i^{\mathrm{I/O}} + \sum_{l \in \mathrm{hp}_i^{\mathrm{I/O}}} \left\lceil \frac{r_i^{\mathrm{I/O},k}}{p_l^{\mathrm{I/O}}} \right\rceil e_l^{\mathrm{I/O}} > \mathrm{sbf}_L(\max_i D_i^{\mathrm{I/O}})$, then Equation 5.16 is verified. Hence, it suffices to compute $\mathrm{sbf}_L(t), \overline{\mathrm{sbf}}_U(t)$ in $[0, \max_i D_i^{\mathrm{I/O}}]$.

Luckily, we do not need to run Algorithm 8 for all values of t in $[0, \max_i D_i^{\mathrm{I/O}}]$. Since functions $E_i^{\max}(t), E_i^{\min}(t)$ are piecewise linear, we can obtain $\overline{\mathrm{sbf}}_U(t)$ by interpolation of a finite number of values as shown in Algorithm 9. The

Algorithm 8 Compute $\overline{\mathrm{sbf}}_U(t)$ for a given t.

1: **procedure** COMPUTEINSTANT($t, \tau_1, \ldots, \tau_N$ ordered by non-increasing α_i)
2: $t_{rem} := t - \sum_{i=1}^{N} E_i^{\min}(t)$
3: $val := \sum_{i=1}^{N}(\delta_i + \alpha_i E_i^{\min}(t))$
4: $i := 1$
5: **while** $t_{rem} > 0$ and $i \leq N$ **do**
6: $x_i = \min\left(t_{rem}, E_i^{\max}(t) - E_i^{\min}(t)\right)$
7: $t_{rem} := t_{rem} - x_i$
8: $val := val + \alpha_i x_i$
9: $i := i + 1$
10: **end while**
11: **return** $(val, i, \{x_1, \ldots, x_N\})$
12: **end procedure**

algorithm returns $\overline{\mathrm{sbf}}_U(t)$ as a set p_{set} of points $\{\ldots, (t', val'), (t'', val''), \ldots\}$, where for $t' \leq t \leq t''$, $\overline{\mathrm{sbf}}_U(t) = val' + (t - t')\frac{val'' - val'}{t'' - t'}$, e.g. $\overline{\mathrm{sbf}}_U(t)$ is the linear interpolation between (t', val') and (t'', val''). The algorithm first computes in Line 4 the set $t^1, \ldots, t^{M-1}$ of angular points of $E_i^{\max}(t), E_i^{\min}(t)$ in interval $[0, \max_i D_i^{\mathrm{I/O}})$, with t^M being the maximum value of interest $\max_i D_i^{\mathrm{I/O}}$; note that this implies that in every open interval (t^j, t^{j+1}), each function $E_i^{\max}(t), E_i^{\min}(t)$ has a constant slope equal to either 0 or 1. In particular, function $\mathrm{EMaxAdd}(l, t^j)$ returns the slope of $E_l^{\max}(t)$ in interval (t^j, t^{j+1}), while $\mathrm{EMinAdd}(l, t^j)$ returns the slope of $E_l^{\min}9(t)$ in interval (t^j, t^{j+1}). For each point t^j, Algorithm 8 is applied to compute the value $val = \overline{\mathrm{sbf}}_U(t^j)$, as well as the index i of the last task for which variable x_i has been assigned a non-zero value. (t^j, val) is then inserted in p_{set} at Line 11.

Finally, the algorithm iterates in Lines 10-26, increasing the current time value t_{cur} starting at $t_{cur} = t^j$, until t_{cur} exceeds the next angular point t^{j+1}. Note that when Algorithm 8 is run to compute the solution to Equations 5.13-5.15 for $t_{cur} = t^j$, variables x_l will be assigned as follows:

$$x_l = E_l^{\max}(t_{cur}) - E_l^{\min}(t_{cur}) \quad \forall l, 1 \leq l \leq i-1, \tag{5.17}$$

$$x_l = 0 \quad \forall l, i < l \leq N, \tag{5.18}$$

$$x_i = t^j - \sum_{l=1}^{N} E_l^{\min}(t_{cur}) - \sum_{l \neq i} x_l = t^j - \sum_{l=1}^{i-1} E_l^{\max}(t_{cur}) - \sum_{l=i}^{N} E_l^{\min}(t_{cur}). \tag{5.19}$$

Now consider the solution $\overline{val}$, with variable assignment $\bar{x}_1, \ldots, \bar{x}_N$, computed by Algorithm 8 for time instant $\bar{t} = t_{cur} + \Delta$, where Δ is a small enough value. Note that as long as $\bar{t} \leq t^{j+1}$, then $E_l^{\max}(\bar{t}) = E_l^{\max}(t_{cur}) + \Delta\mathrm{EmaxAdd}(l, t_{cur})$ and similarly $E_l^{\min}(\bar{t}) = E_l^{\min}(t_{cur}) + \Delta\mathrm{EminAdd}(l, t_{cur})$. Furthermore, define $div \equiv \sum_{l=1}^{i-1} \mathrm{EMaxAdd}(l, t^j) + \sum_{l=i}^{N} \mathrm{EMinAdd}(l, t^j)$ as computed in Line 12. Then based on Equations 5.17-5.19, we

Algorithm 9 Compute $\overline{\mathrm{sbf}}_U(t)$ in $[0, \max_i D_i^{\mathrm{I/O}}]$.

procedure COMPUTEINTERVAL($\max_i D_i^{\mathrm{I/O}}, \tau_1, \ldots, \tau_N$)
  $\forall i, 1 \le i \le N$: compute α_i, δ_i.
  Order $\tau_i, \ldots, \tau_N$ by non-increasing values of α_i.
  Compute $t^1, \ldots, t^{M-1}$ as the set of angular points for any $E_i^{\max}(t), E_i^{\min}(t)$ in $[0, \max_i D_i^{\mathrm{I/O}})$.
  $t^M := \max_i D_i^{\mathrm{I/O}}$
  $p_{set} := \emptyset$
  for $j = 1 \ldots M-1$ **do**
    $(val, i, \{x_1, \ldots, x_N\}) = \mathrm{ComputeInstant}(t^j, \tau_1, \ldots, \tau_N)$
    $t_{cur} := t^j$
    while $t_{cur} < t^{j+1}$ **do**
      add (t_{cur}, val) to p_{set}
      $div := \sum_{l=1}^{i-1} \mathrm{EMaxAdd}(l, t^j) + \sum_{l=i}^{N} \mathrm{EMinAdd}(l, t^j)$
      if $div > 1$ **then**
        $\Delta := x_i/(div-1)$
        $t_{cur} := t_{cur} + \Delta$
        $val := val + \Delta\big(\sum_{l=1}^{i-1} \alpha_l \mathrm{EMaxAdd}(l, t^j) + \sum_{l=i}^{N} \alpha_l \mathrm{EMinAdd}(l, t^j)\big) - \alpha_i x_i$
        $i := i - 1$
      else if $(div = 0$ & $\mathrm{EMaxAdd}(i, t^j) = 0) \mid (div = 1$ & $\mathrm{EMaxAdd}(i, t^j) = 0$ & $\mathrm{EMinAdd}(i, t^j) = 1)$ **then**
        $\Delta := E_i^{\max}(t_{cur}) - E_i^{\min}(t_{cur}) - x_i$
        $t_{cur} := t_{cur} + \Delta$
        $val := val + \alpha_i \Delta$
        $i := i + 1$
      else
        break
      end if
    end while
  end for
  return p_{set}
end procedure

obtain:

$$\bar{x}_l = E_l^{\max}(\bar{t}) - E_l^{\min}(\bar{t}) \quad \forall l, 1 \leq l \leq i-1, \tag{5.20}$$

$$\bar{x}_l = 0 \quad \forall l, i < l \leq N, \tag{5.21}$$

$$\bar{x}_i = \bar{t} - \sum_{l=1}^{i-1} E_l^{\max}(\bar{t}) - \sum_{l=i}^{N} E_l^{\min}(\bar{t}) = \tag{5.22}$$

$$= t^j + \Delta(1-div) - \sum_{l=1}^{i-1} E_l^{\max}(t_{cur}) - \sum_{l=i}^{N} E_l^{\min}(t_{cur}) = x_i + \Delta(1-div), \tag{5.23}$$

$$\begin{aligned}
\overline{val} &= \sum_{l=1}^{N}(\delta_l + \alpha_l E_l^{\min}(\bar{t})) + \sum_{l=1}^{N} \alpha_l \bar{x}_l = \\
&= \sum_{l=1}^{N} \delta_l + \sum_{l=1}^{i-1} \alpha_l E_l^{\max}(\bar{t}) + \sum_{l=i}^{N} \alpha_l E_l^{\min}(\bar{t}) + \alpha_i \bar{x}_i = \\
&= \Big(\sum_{l=1}^{N} \delta_l + \sum_{l=1}^{i-1} \alpha_l E_l^{\max}(t_{cur}) + \sum_{l=i}^{N} \alpha_l E_l^{\min}(t_{cur}) + \alpha_i x_i \Big) + \\
&+ \sum_{l=1}^{i-1} \alpha_l \Delta\text{EMaxAdd}(l, t^j) + \sum_{l=i}^{N} \alpha_l \Delta\text{EMinAdd}(l, t^j) + \alpha_i \Delta(1-div) = \\
&= val + \sum_{l=1}^{i-1} \alpha_l \Delta\text{EMaxAdd}(l, t^j) + \sum_{l=i}^{N} \alpha_l \Delta\text{EMinAdd}(l, t^j) + \alpha_i \Delta(1-div),
\end{aligned} \tag{5.24}$$

and such solution is valid as long as $\Delta \leq t^{j+1} - t_{cur}$ and furthermore Δ is small enough that it holds: $0 \leq \bar{x}_i \leq E_i^{\max}(\bar{t}) - E_i^{\min}(\bar{t})$; note that based on Equation 5.23, the latter condition can be rewritten as:

$$x_i + \Delta(1-div) \geq 0, \tag{5.25}$$

$$x_i + \Delta(1-div) \leq E_i^{\max}(t_{cur}) - E_i^{\min}(t_{cur}) + \big(\text{EMaxAdd}(i, t^j) - \text{EMinAdd}(i, t^j)\big)\Delta. \tag{5.26}$$

Since $\bar{x}_i$ can be increasing or decreasing with Δ based on the value of div, we have to consider several different cases. In all cases, note that Equation 5.24 is linear in Δ, hence as long as the Equation holds in an interval (t', t''), the solution to Equations 5.13-5.15 can be obtained by linear interpolation of $(t', val'), (t'', val'')$.

Case (1): $div > 1$ (Lines 14-17). The constraint in Equation 5.26 is verified for all Δ. Solving Equation 5.25 yields $\Delta \leq x_i/(div-1)$. We have two subcases **(a)** and **(b)**. **(a)** Assume that $x_i/(div-1) < t^{j+1} - t_{cur}$. Then Equations 5.23, 5.24 are valid in the interval $[t_{cur}, t_{cur} + x_i/(div-1)]$. Note that for $\Delta = x_i/(div-1)$, $\bar{x}_i = 0$. Furthermore, we have $\alpha_i \Delta(1-div) = -\alpha_i x_i$. Hence, $\overline{val}$ can be computed as in Line 16, new values for t_{cur}, val are assigned as $t_{cur} := t_{cur} + x_i/(div-1)$, $val := \overline{val}$ and the new point (t_{cur}, val) is inserted in Line 11 at the next iteration. Finally, note that at the next iteration, all variables x_l with $l < i-1$ are still assigned their maximum value $E_l^{\max}(t_{cur}) - E_l^{\min}(t_{cur})$ and all $x_l, l \geq i$ are set to 0. Hence, we can apply the same reasoning as in Equations 5.20-5.24 after setting $i := i-1$ as in Line 17. In particular, note that i can never be assigned the invalid value

0. By contradiction, assume that in the current iteration $i = 1$. Then all $x_l = 0$, meaning that $\bar{t} = \sum_{l=1}^{N} E_l^{\min}(\bar{t})$ according to Equation 5.14; this in turn implies $\sum_{l=1}^{N} \mathrm{EMinAdd}(l, t^j) = 1$, which contradicts $div > 1$. **(b)** Assume that $x_i/(div - 1) \geq t^{j+1} - t_{cur}$. Then Equations 5.23, 5.24 are valid in the interval $[t_{cur}, t^{j+1}]$, the condition of the while loop in Line 10 becomes false and the point (t^j, val) is added to p_{set} in the next iteration of the loop at Line 7.

Case (2): $div = 0$ and $\mathrm{EMaxAdd}(i, t^j) = 0$ (Line 19-22). Note $div = 0$ implies $\mathrm{EMinAdd}(i, t^j) = 0$. The constraint in Equation 5.25 is verified for all Δ. Solving Equation 5.26 yields $\Delta \leq E_i^{\max}(t_{cur}) - E_i^{\min}(t_{cur}) - x_i$. Once again we have two subcases. **(a)** Assume that $E_i^{\max}(t_{cur}) - E_i^{\min}(t_{cur}) - x_i < t^{j+1} - t_{cur}$. Then Equations 5.23, 5.24 are valid in the interval $[t_{cur}, t_{cur} + E_i^{\max}(t_{cur}) - E_i^{\min}(t_{cur}) - x_i]$. Note that for $\Delta = E_i^{\max}(t_{cur}) - E_i^{\min}(t_{cur}) - x_i$, $\bar{x}_i$ is set to the maximum value $E_i^{\max}(\bar{t}) - E_i^{\min}(\bar{t})$. Also note that in Equation 5.24, $\alpha_i \Delta(1 - div) = \alpha_i \Delta$ and $\sum_{l=1}^{i-1} \alpha_l \Delta \mathrm{EMaxAdd}(l, t^j) + \sum_{l=i}^{N} \alpha_l \Delta \mathrm{EMinAdd}(l, t^j) = 0$, hence $\overline{val}$ can be computed as in Line 21. The same considerations as in Case (1a) then apply except that we set $i := i + 1$ since in the next iteration, all variables x_l with $l \leq i$ are assigned their maximum value $E_l^{\max}(t_{cur}) - E_l^{\min}(t_{cur})$ and all $x_l, l > i + 1$ are still set to 0. Note that it is possible to set i to the invalid value $N + 1$; this is covered in Case (6). **(b)** Assume that $E_i^{\max}(t^j) - E_i^{\min}(t^j) - x_i \geq t^{j+1} - t_{cur}$. Then the same considerations as in Case (1b) apply.

Case (3): $div = 1$ and $\mathrm{EMaxAdd}(i, t^j) = 0, \mathrm{EMinAdd}(i, t^j) = 1$ (Line 19-22). As in Case (2), the constraint in Equation 5.25 is verified for all Δ, while solving Equation 5.26 yields $\Delta \leq E_i^{\max}(t_{cur}) - E_i^{\min}(t_{cur}) - x_i$. Furthermore, note that for $\Delta = E_i^{\max}(t_{cur}) - E_i^{\min}(t_{cur}) - x_i$, $\bar{x}_i$ is again set to the maximum value $E_i^{\max}(\bar{t}) - E_i^{\min}(\bar{t})$ and from Equation 5.24 we obtain $\overline{val} = val + \alpha_i \Delta$. Hence, this case is equivalent to Case (2a)-(2b) and can be handled by the same Lines 19-22.

Case (4): $div = 0$ and $\mathrm{EMaxAdd}(i, t^j) = 1$ (Line 24). As in Case (2), note $div = 0$ implies $\mathrm{EMinAdd}(i, t^j) = 0$. Then both Equations 5.25 and 5.26 are verified for all Δ. Hence, as in Case (1b), Equations 5.23, 5.24 are valid in the interval $[t_{cur}, t^{j+1}]$ and the point (t^j, val) is added to p_{set} in the next iteration of the for loop.

Case (5): $div = 1$ and either $\mathrm{EMaxAdd}(i, t^j) = 1$ or $\mathrm{EMinAdd}(i, t^j) = 0$ (Line 24). Then again both Equations 5.25 and 5.26 are verified for all Δ, hence this case is equivalent to Case (4).

Case (6): $i > N$ (Line 24). This can only happen if in the previous iteration either Case (2a) or (3a) is executed with $i = N$. Note that in both cases, in the current iteration $div = \sum_{l=1}^{N} \mathrm{EMaxAdd}(l, t^j) = 0$, hence Line 24 is executed. We show that the assignment where each variable is maximal, e.g. $x_l = E_l^{\max}(\bar{t}) - E_l^{\min}(\bar{t})$, is optimal for all $\bar{t}$ in the interval $[t_{cur}, t^{j+1}]$. Equation 5.14 is equivalent to: $\sum_{l=1}^{N} \left(E_l^{\max}(\bar{t}) - E_l^{\min}(\bar{t})\right) \leq \bar{t} - \sum_{l=1}^{N} E_l^{\min}(\bar{t})$, which is in turn equivalent to: $\sum_{l=1}^{N} E_l^{\max}(\bar{t}) \leq \bar{t}$. But since $\mathrm{EMaxAdd}(l, t^j) = 0$ for all l and furthermore at t_{cur} it must hold $\sum_{l=1}^{N} E_l^{\max}(t_{cur}) \leq t_{cur}$, Equation 5.14 is verified for all $\bar{t}$ in $[t_{cur}, t^{j+1}]$. Hence, the assignment $x_l = E_l^{\max}(\bar{t}) - E_l^{\min}(\bar{t})$ must result in the optimum. Computing $\overline{val}$ according to Equation 5.24 results in $\overline{val} = val$, hence $\overline{\mathrm{sbf}}_U(t^j)$ is constant between point (t_{cur}, val) and point (t^j, val), which is added to p_{set} in the next iteration of the for loop.

We can now prove our main theorem.

Theorem 38 *A correct upper bound* $\overline{\mathrm{sbf}}_U(t)$ *to* $\overline{\mathrm{sbf}}(t)$ *in the interval* $[0, \max_i D_i^{\mathrm{I/O}}]$ *can be computed by linear interpolation of the point set returned by Algorithm 9.*

Proof.

Based on the discussion above, whenever Algorithm 9 inserts two consecutive points $(t', val'), (t'', val'')$ in p_{set}, $\overline{\mathrm{sbf}}_U(t)$ can be computed by linear approximation of $(t', val'), (t'', val'')$ for all t in the interval $[t', t'']$. Since furthermore points are inserted for the beginning and end of the interval $[0, \max_i D_i^{\mathrm{I/O}}]$, to conclude the proof it remains to show that the algorithm terminates; this is not obvious, since in the while loop in Lines 10-26, Δ can be set to 0 and furthermore i can be either incremented or decremented. We show that if i is incremented in the first iteration of the while loop, then it can not be decremented in further iterations; similarly, if i is decremented in the first iteration, it can not be incremented afterwards. Since $i \geq 1$ by Case (1a), and furthermore the while loop is terminated whenever $i > N$, it follows that the loop is executed at most $N + 1$ times.

By contradiction, assume that Lines 14-17 are executed in one iteration, and Lines 19-22 in the following. Note that if i is incremented or decremented by 1, the value of div can only change by 1. Hence, it must hold $div = 2$ in the first iteration and $div = 1$ in the second. However, this implies $\mathrm{EMaxAdd}(i, t^j) = 1$ in the second iteration, hence Lines 19-22 can not be executed. Similarly, assume that Lines 19-22 are executed in one iteration, and Lines 14-17 in the following; it must hold $div = 1$ in the first iteration and $div = 2$ in the second. This implies $\mathrm{EMinAdd(i, t^j)} = 0$ in the first iteration, hence Lines 19-22 can not be executed. □

It remains to discuss the computational complexity of Algorithm 9. Note that in a time window of length t, the number of angular points for $E_i^{\max}(t)$ or $E_i^{\min}(t)$ are not more than $2 + 2\lceil \frac{t}{p_i} \rceil$. Hence, an upper bound to the number of iterations of the for loop at Line 7 can be computed as $N\Big(4 + 4\lceil \frac{\max_i D_i^{\mathrm{I/O}}}{\min_i p_i} \rceil\Big) + 1$, where the 1 term accounts for $t^M = \max_i D_i^{\mathrm{I/O}}$. Lines 8-9 can be executed in $O(N)$. Based on the proof of Theorem 38, the while loop at Line 10 is repeated at most $N + 1$ times. Finally, note that Lines 11-25 can be optimized to run in constant time rather than linear time. This is because the value of i changes by 1 between successive iterations. Hence, the summations in Lines 12, 16 can be computed based on their values at the previous step in constant time. In conclusion, Algorithm 9 has a pseudo-polynomial complexity of $O\big(N^2 \max_i D_i^{\mathrm{I/O}} / \min_i p_i\big)$.

5.2.4 Evaluation

In order to verify the validity and practicality of PREM, we implemented the key components of the system. In this section, we describe our evaluation, first introducing the new hardware components followed by the corresponding software driver and OS calibration effort. We then discuss our compiler implementation and analyse its effectiveness on a DES benchmark. Finally, using synthetic tasks we measure the effectiveness of the PREM system as a function of cache stall time, and show traces of PREM when running on COTS hardware.

PREM Hardware Components

To support PREM, a few modifications were performed to the real-time bridge and peripheral scheduler design described in Section 2.1. First of all, each real-time bridge communicates with the peripheral scheduler using three wires: a `data_ready` and `block` wire as before, plus a new `interrupt_block` wire. The `interrupt_block` signal instructs the real-time bridge to not further raise interrupts, and is used to block all peripheral interrupts during predictable interrupts.

PREM requires the CPU and peripherals to coordinate their access to main memory. The peripheral scheduler is connected to the PCIe bus, and exposes a set of registers accessible from the main CPU. In the `configuration` register, constant parameters such as the maximum cache write-back time are stored. Writing a value to the `yield` register indicates that the CPU will not access main memory for the given amount of time, and I/O peripherals should be allowed to read to and write from RAM. The value written to the `yield` register contains a 14 bit unsigned integer indicating the number of microseconds to permit peripheral traffic with main memory, as described in Section 5.2.1. The CPU can also use the `yield` register to allow interrupts to be raised by peripherals. Another 14 bit unsigned integer indicates the number of microseconds to allow peripheral interrupts, and a 3 bit interrupt mask selects which peripherals should be allowed to raise the interrupts. In this way, different CPU tasks can predictably service interrupts from different I/O peripherals.

Software Evaluation

The software effort required to implement PREM execution involves two aspects: (1) creating the drivers which control the custom PREM hardware discussed in the previous section, and (2) calibrating the OS to eliminate undesired execution interference. We now discuss these, in order.

The two custom hardware components, the real-time bridge and the peripheral scheduler, each require a software driver to be controller from the main CPU. Additionally, each peripheral requires a driver running on the real-time bridge's CPU to control the COTS peripheral. The driver for the peripheral scheduler is straightforward, mapping the bus addresses corresponding to the exposed registers to user space where a PREM-compiled process can access them. The driver for each real-time bridge is more difficult, since each unique COTS peripheral requires a unique driver. However, since in our implementation both the main CPU and the real-time bridge's CPU are running Linux (version 2.6.31), we can reuse existing, thoroughly tested Linux drivers to drastically reduce the driver creation effort [9]. The presence of a real-time bridge is not apparent in user space, and software programs using the COTS peripherals require no modification.

For our experiments, we use a Intel Q6700 CPU with a 975X system controller; as in all previous experiments, we slowed down the CPU speed, in this case to a frequency of 1Ghz obtaining a measured memory bandwidth of 1.8Ghz/s. We also disable the speculative CPU HW prefetcher since it negatively impacts the predictability of any real-time task. The Q6700 has four CPU cores and each pair of cores shares a common level 2 (last level) cache. Each

Data size	4K	8K	32K	128K	512K	1M
Compatible	138	254	954	3780	15k	31k
Predictable	2	2	4	2	1	81

Table 5.2: DES benchmark.

cache is 16-associative with a total size of $B = 4$ Mbytes and a line size of $L = 64$ bytes. Since we use a PC platform running a COTS Linux operating system, there are many potential sources of timing noise, such as interrupts, kernel threads, and other processes, which must be removed for our measurements to be meaningful. For this reason, in order to emulate at our best a typical uni-processor embedded real-time platform, we divided the 4 cores in two partitions. The *system partition*, running on the first pair of cores, receives all interrupts for non-critical devices (ex: the keyboard) and runs all the system activities and non real-time processes (ex: the shell we use to run the experiments). The *real-time partition* runs on the second pair of cores. One core in the real-time partition runs our real-time tasks together with the drivers for real-time bridges and the peripheral scheduler; the other core is not used. Note that the cores of the system partition can still produce a small amount of unscheduled bus and main memory accesses, or raise rare inter-processor interrupts (IPI) that can not be easily prevented. However, in our experiments we found these sources of noise to be negligible. Finally, to solve the paging issue detailed in Section 5.2.1, we used a large, 4MB page size, just for the real-time tasks, using the HugeTLB feature of the Linux kernel for large page support.

Compiler Evaluation

We built the PREM real-time compiler prototype using the LLVM Compiler Infrastructure [52], targeting the compilation of C code. LLVM was extended by writing self-contained analysis and transformation passes, which were then loaded into the compiler.

In the current PREM real-time compiler prototype, we rely on the programmer to partition the task into predictable and compatible intervals . The partitioning is done by putting each predictable interval into its own function. The beginning and end of the scheduling interval correspond to the entry and exit of the function, and the start of execution phase (the end of memory access phase) is manually marked by the programmer. We assume that non-local data accessed during a predictable interval exists in continuous memory spaces, which can be prefetched by a set of `PREFETCH_DATA(start_address, size)` macros that must be placed by the programmer during the memory phase. The implementation of this macro does the actual prefetching of the data into level 2 cache by prefetching every cache line in the given range with the i386 `prefetcht2` instruction. After the memory phase, the programmer adds a `STARTEXECUTION(wcet)` macro to indicate the beginning of the execution phase. This macro measures the amount of time remaining in the predictable interval using the CPU performance counter, and writes the time remaining to the `yield` register in the peripheral scheduler.

All remaining operations needed to transform the interval are performed by a new LLVM function pass. The pass

iterates over all the functions in a compilation unit. When a function representing a predictable interval is found, the pass performs code transformation. First, our transform inlines called functions using preexisting LLVM inlining functions. This ensures that there are only a single stack frame and segment of code that need to be prefetched into the cache. Second, our transform inserts code to read the CPU performance counter at the beginning of the interval and save the current time. Third, it inserts code to prefetch the stack frame and function arguments. Bringing the stack frame into the cache is done by inserting instructions into the program to fetch the stack pointer and frame pointer. Code is then inserted to prefetch the memory between the stack pointer and slightly beyond the frame pointer (to include function arguments) using the `prefetcht2` instruction. Fourth, the transform prefetches the code of the function. This is done by transforming the program so that the function is placed within a unique ELF section. We then use a linker script to define variables pointing to the beginning and end of this unique ELF section. The compiler then adds code that prefetches the memory inside the ELF section. Finally, the pass identifies all return instructions inside the predictable interval function and adds a special function epilog before them. The epilog performs interval length enforcement by looping until the performance counter reaches the worst-case cycle count based on the time value saved at the beginning of the interval. It may also enable peripheral interrupts by writing the worst-case interrupt processing time to the peripheral scheduler's `yield` register.

To verify the correctness of the PREM real-time compiler prototype and to test its applicability, we used LLVM to compile a DES cypher benchmark. The DES benchmark was selected because it represents a typical real-time data flow application. The benchmark comprises one scheduling interval which encrypts a variable amount of data. We compiled it as both a predictable and a compatible interval (e.g. with and without prefetching), and measured number of cache misses with a performance counter. Adapting the interval required no modification to any cypher functions and a total of 11 `PREFETCH_DATA` macros.

Results are shown in Table 5.2 in terms of the number of cache misses suffered in the execution phase of the predictable interval (after prefetching), and in the entire compatible interval. Data size is in bytes. The compatible interval suffers an excessive number of cache misses, which increases roughly proportionally with the amount of processed data. Conversely, the execution phase of the predictable interval has almost zero cache misses, only suffering a small increase when large amounts of data are being processed. The reason the number of cache misses is not zero is that the Q6700 CPU core used in our experiments uses a random cache replacement policy, meaning that with more than one contiguous memory region the probability of self-eviction is non-zero. In all the following experiments, we observed that the number of self-evictions is typically so small that it can be considered negligible.

WCET Experiments with Synthetic Tasks

In this section, we evaluate the effects of PREM on the execution time of a task. To quickly explore different execution parameters, we developed two synthetic applications. In our `linear_access` application, each scheduling interval operates on a 256-kilobyte global data structure. Data is accessed sequentially, and we vary the amount of computation

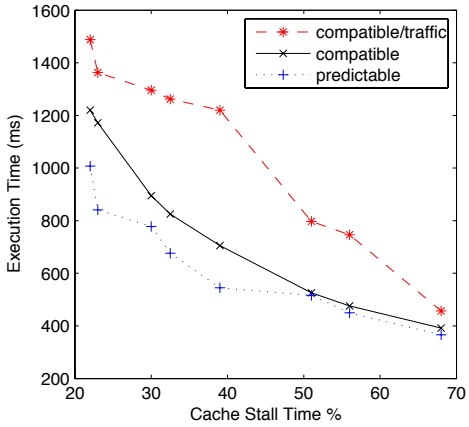


Figure 5.16: `random_access`

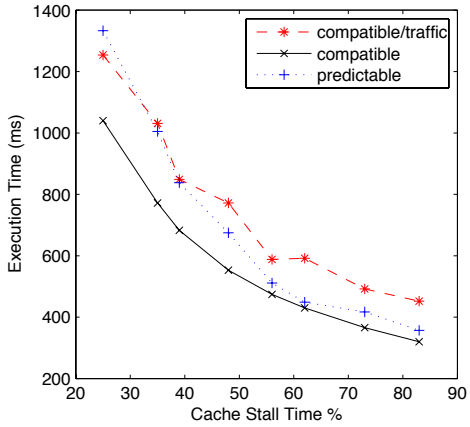


Figure 5.17: `linear_access`

performed between memory references. The `random_access` application is similar, except that references inside the data structure are nonsequential. For each application, we measured the execution time after compiling the program in two ways: into predictable intervals which prefetch the accessed memory, and into standard, compatible intervals. For each type of compilation, we ran the experiment in two ways, with and without I/O traffic transmitted by an 8-lane PCIe peripheral with a measured throughput of 1.2Gbytes/s. In the case of compatible intervals, we transmitted traffic during the entire interval to mirror the worst case according to the traditional execution model.

Figures 5.16 and 5.17 show the observed worst case execution time for any scheduling interval as a function of the cache stall time of the application, averaged over 10 runs. The cache stall time represents the percentage of time required to fetch cache lines out of an entire compatible interval, assuming a fixed (best-case) fetch time based on the maximum measured main-memory throughput. Only a single line is shown for predictable intervals because experiments confirmed that injecting traffic during the execution phase does not increase execution time. In all cases, the computation time decreases with an increase in stall time. This is because stall time is controlled by varying the amount of computation between memory references. Furthermore, execution times should not be compared between the two figures because the two applications execute different code.

In the `random_access` case, predictable intervals outperform compatible intervals (without peripheral traffic) by up to 28%, depending on the cache stall time. We believe this effect is primarily due to the behavior of DRAM main memory. Specifically, accesses to adjacent addresses can be served quicker in burst mode than accesses to random addresses. Thus, we can decrease the execution time by loading all the accessed memory into cache, in order, at the beginning of each predictable interval. Furthermore, note that transmitting peripheral traffic during a compatible interval can increase execution time by more than 60% in the worst case. In Figure 5.17, predictable intervals perform worse than compatible intervals (without peripheral traffic). We believe this is mainly due to out-of-order execution in the Q6700 core. In compatible intervals, while the core performs a cache fetch, instructions in the pipeline that do

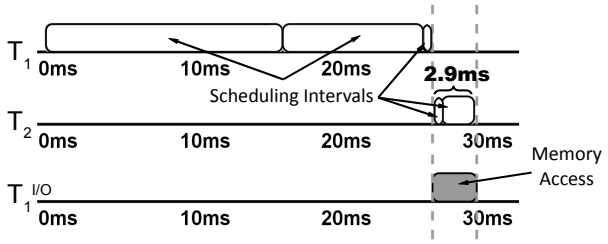


Figure 5.18: Unscheduled trace without PREM.

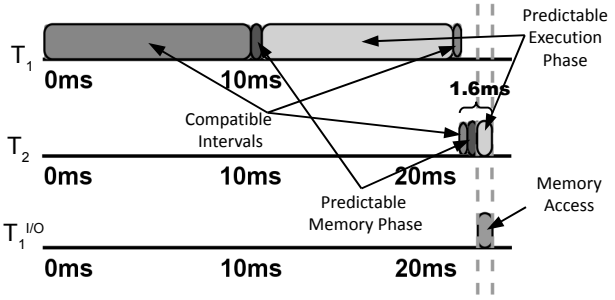


Figure 5.19: Scheduled trace using PREM.

not depend on the fetched data can continue to execute. When performing linear accesses, fetches require less time and this effect is magnified. Furthermore, the gain in execution time for the case with peripheral traffic is decreased: this occurs because bursting data on the memory bus reduces the amount of blocking time suffered by a task due to peripheral interference (this effect has been previously analyzed in detail [72]). In practice, we expect the effect of PREM on an application's execution time to be between the two figures, based on the specific memory access pattern.

System-wide Coscheduling Traces

We now present execution traces of the implemented system which demonstrate the advantage of the PREM coscheduling approach. The traces are obtained by using the peripheral scheduler as a logic analyzer for the various signals which are being sent to or from the real-time bridges, `data_block`, `data_ready`, and `interrupt_block`. Additionally, the peripheral scheduler has a `trace` register which allows timestamped trace information to be recorded with a one microsecond resolution when instructed by the main CPU, such as at the start and end of an execution interval. An execution trace is shown for a task running the traditional COTS execution model in Figure 5.18, and the same task running within the PREM model is shown in Figure 5.19[3].

In the first trace (Figure 5.18), although the execution is divided into unpreemptable intervals, there is no memory phase prefetch or constant execution time guarantees. When the scheduling intervals of task T_1 finish executing, an I/O peripheral begins to access main memory, which may happen if T_1 had written output data into RAM. Task T_2 then executes, suffering cache misses that compete for main memory bandwidth with the I/O peripheral. Due to the cold cache and peripheral interference, the execution time of T_2 grows from 0.5 ms (the execution time with warm cache and no peripheral I/O), to 2.9ms as shown in the figure, an increase of about 600%.

In the second trace (Figure 5.19), the system executes according to the PREM execution model, where peripherals only access the bus when permitted by the peripheral scheduler. The predictable interval is divided into a memory phase and an execution phase. Instead of competing for main memory access, task T_2 gets contentionless access to

[3]The (compatible) intervals at the end of T_1 and at the start of T_2 were measured as 0.107ms and 0.009ms, respectively, and have been exaggerated in the figures to be visible.

main memory during the memory phase. After all to-be-accessed data is loaded into the cache, the execution phase begins which incurs no cache misses, and the peripheral is allowed to access the data in main memory. The constant execution time for the predictable interval in the PREM execution model is 1.6ms, which is significantly lower than the worst-case observed for the unscheduled trace (and is about the same as the execution time of the scheduling interval of T_2 in the unscheduled trace with a cold cache and no peripheral traffic).

5.3 Related Work

Several solutions have been proposed in prior real-time research to address different sources of unpredictability in COTS components, including real-time handling of peripheral drivers and real-time compilation. For peripheral drivers, Facchinetti et al. [32] proposed using a non-preemptive interrupt server to better support the reusing of legacy drivers. Additionally, analysis can be done to model worst-case temporal interference caused by device drivers [54]. For real-time compilation, a tight coupling between compiler and worst-case execution time (WCET) analyzer can optimize a program's WCET [33]. Alternatively, a compiler-based approach can provide predictable paging [79]. All these works attempt to analyze or control a single resource, and obtain safe bounds that are often highly pessimistic. Instead, PREM is based on a global coschedule of all relevant system resources.

Instead of using COTS components, other researchers have discussed new architectural solutions that can greatly increase system predictability by removing significant sources of interference. Instead of a standard cache-based architecture, a real-time scratchpad architecture can be used to provide predictable access time to main memory [104]. The Precision Time (PRET) machine [28] promises to simultaneously deliver high computational performance together with cycle-accurate estimation of program execution time. While our PREM execution model borrows some ideas from these works, it exhibits one key difference: our model can be applied to existing COTS-based systems, without requiring significant architectural redesign. This approach allows PREM to leverage the advantage of the economy of scale of COTS systems, and support the progressive migration of legacy systems.

5.4 Future Work

Our evaluation of the PRedictable Execution Model has shown that by enforcing a high-level coschedule among CPU tasks and peripherals, we can greatly reduce or outright eliminate low-level contention for shared resource accesses. We plan to further develop our solution in two main directions. First, our programming model currently relies on a set of very restrictive assumptions. We would like to study extensions to our compiler infrastructure to lift some of the more restrictive code assumptions and increase the level of automation, in particular by automatically analyzing memory references and separation points between scheduling intervals. The goal is to show that automatic compilation, or semi-automatic compilation with the use of very few code annotations, is possible for a larger set of relevant benchmarks.

Second, so far we have proven the applicability of PREM only to single core systems. However, contention for shared resources becomes more severe as the number of active components increases. In particular, worst-case execution time can greatly degrade in multicore systems as shown in Section 4.4. Hence, our final goal is to apply PREM to the scheduling of tasks and I/O flows in multicores. Since PREM can make the system contentionless, we predict that the benefits of our approach will become even more significant in this new context. At the same time, the extension to multicore is not trivial, since it requires both new control mechanisms (in particular, task schedulers on each core must be accurately synchronized among each other and with the peripheral scheduler) and new schedulability analyses (intuitively, tasks must be executed in such a way that no two cores are inside a memory phase at the same time).

Chapter 6

Conclusions

Embedded systems are experiencing a shift towards progressively more integrated architectures. Features such as heterogeneous coprocessors, DMA peripherals, multi-level caches and superscalar execution are employed to satisfy ever growing performance requirements. This increase in system complexity is posing extraordinary challenges to the integration phase of safety-critical embedded systems. New architectural solutions and design methodologies are clearly required to reduce the complexity of system design and verification while still meeting all performance requirements.

The solution proposed in this work is based on the concept of hardware control abstraction. A set of hardware wrappers is built around each unverified COTS component to control its resource access requests. Functional and timing isolation guarantees are provided by monitoring interactions with the rest of the system and imposing a high-level schedule for access to shared resources. When components can not be controlled, an analysis of resource usage patterns can be used to derive bounds on contention induced delay.

I believe that the described work and experimental evaluations have shown that this methodology can be effective at safely integrating COTS components in mixed-criticality systems. Our case study in hardware monitoring of COTS peripherals in Section 3.5 has been remarkable in identifying errors caused by driver faults. Based on initial testing in Section 5.2.4, PREM in conjunction with our I/O management system has been able to significantly improve the timing predictability of both task and peripheral execution, with no or small overhead compared to a traditional execution model.

At the same time, our work has uncovered new challenges that must be addressed to successfully transition our methodology to all aspects of system design. First of all, designing high-performance hardware control abstractions is not trivial. The main reason, as apparent in the case of the real-time bridge described in Section 2.1, is that the implementation of control mechanisms suffers from the inherent complexity of COTS interfaces such as the PCI bus. A second related issue is that some specific architectural choices can impact either the performance of the control mechanisms or the pessimism of analysis bounds in an extremely negative way. Random cache replacement policy is

one such case, as detailed in Section 5.2.1.

To overcome these issues and facilitate technology transition in the industry, a mixed approach is likely to be effective: together with the use of control abstractions, suitable suggestions should be provided to device manufacturers to avoid architectural choices that have catastrophic effects on the predictability of the system. The SoC market is especially attractive in this sense: most manufacturers create platforms by choosing available IPs for complex modules such as CPU and then customizing the architecture of the interconnections. While the manufacturer is typically unwilling to redesign an IP, changes in the interconnection and types of selected components are feasible propositions. For example, modern multicore SoC such as the Freescale QorIQ platform [37], which is marketed for avionic applications, tend to implement a level-3 cache shared among all computational cores. However, the delay analysis of Section 4.4 clearly implies that additional levels of shared resource are bad for predictability, because they increase worst-case access contention. Simply splitting level-3 cache into separate private cache banks, plus eventually a shared bank for shared logical data, would go a long way towards increasing the predictability of the system and simplifying both delay analysis and the design of control mechanisms such as PREM.

Finally, there are many areas in which our proposed solutions could be improved. The inability to express a physical concept of time in our formal monitoring properties creates a clear fracture between the specification of functional and timing properties for the system. This limits our ability to automatically extract formal certificates from a high-level system description, which is a highly desirable characteristic of an abstraction-based design process; as described in Section 3.7, extending HardwareMOP to cover timed automata specification is crucial to bridge the gap. While our multicore delay analysis in Section 4.4 makes reasonably realistic assumptions about the arbitration for access to shared resources, it still simplifies the operation of the memory controller quite a bit. In particular, neither our delay analysis nor PREM can exploit the ability of a modern DRAM controller to access different memory banks in parallel. Furthermore, as detailed in Section 5.4, we have so far demonstrated the use of PREM only on single core systems, and more work is required to adapt the model to multicores. Finally, in this work we have focused our attention on single computational nodes. Complex embedded systems such as aircrafts are composed of tens or even hundreds of computational nodes. While some of the ideas developed in this work could be applied to the control of distributed elements such as an embedded local networks, different analysis methodologies are required to deal with the intrinsic asynchronicity in these systems. We will hopefully be able to address some of these problems as part of our future work.

Bibliography

[1] *Medical Plug and Play Program*. http://mdpnp.org/.

[2] Aeronautical Radio Inc. *ARINC 653 Specification*. http://www.arinc.com/.

[3] B. Akesson, K. Goossens, and M. Ringhofer. Predator: a predictable sdram memory controller. In *Proc. of the 5th IEEE/ACM inter. conference on HW/SW codesign and system synthesis*, 2007.

[4] R. Alur and D.L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.

[5] M. Alvarez, E. Salami, A. Ramirez, and M. Valero. A performance characterization of high definition digital video decoding using h.264/avc. In *Proc. of the IEEE International Workload Characterization Symposium*, Oct 2005.

[6] P. Avgustinov, A. Christensen, L. Hendren, S. Kuzins, J. Lhotak, O. Lhotak, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. ABC: an extensible AspectJ compiler. In *Proc. of the ACM Conf. on Aspect-oriented software development (ASOD'05)*, pages 87–98, 2005.

[7] P. Avgustinov, J. Tibble, and O. de Moor. Making trace monitors feasible. In *Proc. of the ACM Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'07)*, pages 589–608, 2007.

[8] L. Sha B. Sprunt, J.P. Lehoczky. Scheduling sporadic and aperiodic events in a hard real-time system. Technical report, CMU, 1989.

[9] S. Bak, E. Betti, R. Pellizzoni, M. Caccamo, and L. Sha. Real-time control of I/O COTS peripherals for embedded systems. In *Proc. of the 30th IEEE Real-Time Systems Symposium*, Washington DC, Dec 2009.

[10] S. Bak, D. Chivukula, O. Adekunle, M. Sun, M. Caccamo, and L. Sha. The system-level simplex architecture for improved real-time embedded system safety. In *Proceedings of the IEEE RTAS*, 2009.

[11] T. Baker. Lessons learned integrating COTS into systems. In *Proceedings of the First International Conference on COTS-Based Software Systems (ICCBSS 2002)*, Orlando, Florida, Feb 2002.

[12] H. Barringer, A. Goldberg, K. Havelund, and K. Sen. Rule-based runtime verification. In *Int. Conf. on Verification, Model Checking, and Abstract Interpretation (VMCAI'04)*, pages 277–306, 2004.

[13] S. Baruah. The limited-preemption uniprocessor scheduling of sporadic task systems. In *Proceedings of the EuroMicro Conference on Real-Time Systems*, pages 137–144, Palma de Mallorca, Spain, July 2005.

[14] S.K. Baruah, L.E. Rosier, and R.R. Howell. Algorithms and complexity concerning the preemptive scheduling of periodic real-time tasks on one processor. *The Journal of Real-Time Systems*, 2, 1990.

[15] Pam Binns, Matt Englehart, Mike Jackson, and Steve Vestal. Domain Specific Software Architectures for Guidance, Navigation and Control. *International Journal of Software Engineering and Knowledge Engineering*, 6(2):201–227, 1996.

[16] J.-Y. Le Boudec and P. Thiran. *Network Calculus: A Theory of Deterministic Queuing Systems for the Internet*. Springer, LNCS, 2001.

[17] B. D. Bui, M. Caccamo, L. Sha, and J. Martinez. Design and evaluation of a cache partitioned environment for real-time embedded systems. In *Proceedings of the 14th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, KaoHsiung, Taiwan, 2008.

[18] BusMOP webpage. `http://fsl.cs.uiuc.edu/BusMOP`.

[19] G. Buttazzo. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications, Second Edition*. Kluwer Academic Publishers, Boston, 2004.

[20] M. Caccamo, L. Y. Zhang, L. Sha, and G. Buttazzo. An implicit prioritized access protocol for wireless sensor networks. In *Proceedings of the IEEE RTSS*, Dec 2002.

[21] S. Chakraborty, S. Kunzli, and L. Thiele. A general framework for analysing system properties in platform-based embedded system design. In *Proc. of 6th Design, Automation and Test in Europe (DATE)*, Munich, Germany, 2003.

[22] F. Chen and G. Roşu. MOP: An Efficient and Generic Runtime Verification Framework. In *Proc. of the ACM OOPSLA*, pages 569–588, 2007.

[23] T. Chen, R. Raghavan, J. Dale, and E. Iwata. Cell broadband engine architecture and its first implementation: A performance view. Technical report, IBM Research, 2005.

[24] P. Cumming. The TI OMAP platform approach to SoCs. In *Surviving the SoC revolution: A guide to platform-based design*. Kluwer, 1999.

[25] D. Drusinksky. Temporal rover, 1997-2007.

[26] E.A. Emerson. *Handbook of Theoretical Computer Science*. MIT Press, 1990. Chapter 16: Temporal and modal logic.

[27] Eagle Technology. *PCI 703 Series User's Manual*. `http://www.eagledaq.com/display_product_36.htm`.

[28] S. A. Edwards and E. A. Lee. The case for the precision timed (pret) machine. In *DAC '07: Proc. of the 44th annual Design Automation Conference*, 2007.

[29] D. Abramson et al. Intel virtualization technology for directed i/o. *Intel Technology Journal*, 10(03), Aug 2006.

[30] F. Balarin et al. Metropolis: An integrated electronic system design environment. *IEEE Computers*, 36(4):45–52, 2003.

[31] M. Clavel et al. The maude 2.0 system. In *Proc. Rewriting Techniques and Applications*, 2003.

[32] T. Facchinetti, G. Buttazzo, M. Marinoni, and G. Guidi. Non-preemptive interrupt scheduling for safe reuse of legacy drivers in real-time systems. In *ECRTS '05: Proc. of the 17th Euromicro Conf. on Real-Time Systems*, pages 98–105, 2005.

[33] H. Falk, P. Lokuciejewski, and H. Theiling. Design of a wcet-aware c compiler. In *ESTMED '06: Proc. of the 2006 IEEE/ACM/IFIP Workshop on Embedded Systems for Real Time Multimedia*, pages 121–126, 2006.

[34] P.H. Feiler, B.A. Lewis, and S. Vestal. The SAE Architecture Analysis & Design Language (AADL): A Standard for Engineering Performance Critical Systems. In *Proc, of the 2006 IEEE Conference on Computer Aided Control Systems Design*, Oct. 2006.

[35] C. Ferdinand, F. Martin, and R. Wilhelm. Applying compiler techniques to cache behavior prediction. In *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers and Tools for Real-Time Systems*, 1997.

[36] FFMPEG project. *libavcodec multimedia library*. `http://ffmpeg.mplayerhq.hu/`.

[37] Freescale Semiconductor. *QorIQ Communications Platforms*. http://www.freescale.com/webapp/sps/site/homepage.jsp?code=QORIQ_HOME.

[38] Doug Gibbs. Measuring treck tcp/ip performance using the xps locallink temac in an embedded processor system. www.xilinx.com/ support/ documentation/ application_notes/ xapp1043.pdf, 2008.

[39] K. Goossens, J. Dielissen, and A. Radulescu. thereal network on chip: Concepts, architectures, and implementations. *IEEE Design and Test*, 22(5):414–421, 2005.

[40] D. Grund and J. Reineke. Precise and efficient FIFO-replacement analysis based on static phase detection. In *Proceedings of the 22nd Euromicro Conference on Real-Time Systems (ECRTS)*, Brussels, Belgium, July 2010.

[41] N. Guan, M. Stigge, W. Yi, and G. Yu. Cache-aware scheduling and analysis for multi-cores. In *EMSOFT*, October 2009.

[42] K. Havelund and G. Rosu. Monitoring Java programs with Java pathexplorer. In *Proc. First Workshop on Runtime Verification*, 2001.

[43] K. Hoyme and K. Driscoll. Safebus(tm). *IEEE Aerospace Electronics and Systems Magazine*, pages 34–39, Mar 1993.

[44] Tay-Yi Huang, Jane W. S. Liu, and Jen-Yao Chung. Allowing cycle-stealing direct memory access i/o concurrent with hard-real-time programs. In *Int. Conf. on Parallel and Distributed Systems*, Tokyo, 1996.

[45] J. Hugues, B. Zalila, L. Pautet, and F. Kordon. From the prototype to the final embedded system using the Ocarina AADL tool suite. *ACM Trans. on Emb. Computing Sys.*, 7(4):1–25, 2008.

[46] IBM. *Processor Local Bus Specification*. http://www-01.ibm.com/chips/techlib/techlib.nsf/techdocs/3BBB27E5BCC165BA87256A2B0064FFB4.

[47] Intel Corp. *Intel 64 and IA-32 Architectures Software Developer's Manual*, February 2008.

[48] G. Karsai, J. Sztipanovits, A. Ledeczi, and T. Bapty. Model-integrated development of embedded software. *Proceedings of the IEEE*, 91(1):145–164, 2003.

[49] H. Kellerer, U. Pferschy, and D. Pisinger. *Knapsack Problems*. Springer, 2004.

[50] M. Kim, M. Viswanathan, S. Kannan, I. Lee, and O. Sokolsky. Java-mac: A run-time assurance approach for Java programs. *Formal Methods in System Design*, 24(2):129–155, 2004.

[51] D. Knuth. Backus normal form vs. backus naur form. *Communications of the ACM*, 7(12):735–736, 1964.

[52] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis and transformation. In *Proc. of the International Symposium of Code Generation and Optimization*, San Jose, CA, USA, Mar 2004.

[53] J. Lehoczky, L. Sha, and Y. Ding. The rate monotonic scheduling algorithm: Exact characterization and average case behavior. In *Proceedings of the IEEE Real-Time Systems Symposium*, December 1989.

[54] M. Lewandowski, M. Stanovich, T. Baker, K. Gopalan, and A. Wang. Modeling device driver effects in real-time schedulability: Study of a network driver. In *Proc. of the* 13^{th} *IEEE Real Time Application Symposium*, Apr 2007.

[55] Y. Li, V. Suhendra, Y. Liang, T. Mitra, and A. Roychoudhury. Timing analysis of concurrent programs running on shared cache multi-cores. In *RTSS*, 2009.

[56] B. Lickly, I. Liu, S. Kim, H. Patel, S. Edwards, and E. Lee. Predictable programming on a precision timed architecture. In *Proceedings of International Conference on Compilers, Architecture, and Synthesis from Embedded Systems*, Oct 2008.

[57] J. Liedtke, H. Hartig, and M. Hohmuth. OS-controlled cache predictability for real-time systems. In *Proceedings of the 3rd IEEE Real-Time Technology and Applications Symposium (RTAS)*, 1997.

[58] C.L. Liu and J.W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the Association for Computing Machinery*, 20(1), 1973.

[59] J. W. S. Liu. *Real-Time Systems*. Prentice Hall, 2000.

[60] H. Lu and A. Forin. The design and implementation of p2v, an architecture for zero-overhead online verification of software programs. Technical Report MSR-TR-2007-99, Microsoft Research, 2007.

[61] M. Sipser. *Introduction to the Theory of Computation*. PWS Publishing, 1996. Chapter 1: Regular Languages.

[62] M. Martin, B. Livshits, and M. Lam. Finding application errors and security flaws using PQL: a program query language. In *Proc. of the ACM Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'05)*, pages 365–383, 2005.

[63] Marvell. *Discovery II PowerPC System Controller MV64360 Specifications*. available at http://www.marvell.com/.

[64] F. Mueller. Timing analysis for instruction caches. *Real Time Systems Journal*, 18(2/3):272–282, May 2000.

[65] M. Y. Nam, R. Pellizzoni, R. M. Bradford, and L. Sha. ASIIST: Application specic I/O integration support tool for real-time bus architecture designs. In *Proceedings of the IEEE ICECCS*, Potsdam, Germany, 2009.

[66] NXP Semiconductors. *Philips Nexperia Digital Video Platform*. http://www.nxp.com.

[67] S. Oikawa and R. Rajkumar. Linux/rk: a portable resource kernel in linux. In *Proceedings of the* 19^{th} *IEEE Real-Time System Symposium*, Madrid, Spain, December 1998.

[68] M Paolieri, E Quinones, F. J. Cazorla, and M. Valero. An analyzable memory controller for hard real-time CMPs. *IEEE Embedded System Letter*, 1(4), Dec 2009.

[69] PCI SIG. *Conventional PCI 3.0, PCI-X 2.0 and PCIe 2.0 Specifications*. `http://www.pcisig.com`.

[70] R. Pellizzoni, B.D. Bui, M. Caccamo, and Lui Sha. Coscheduling of CPU and I/O transactions in COTS-based embedded systems. In *Real-Time Systems Symposium, 2008*, pages 221–231, 30 2008-Dec. 3 2008.

[71] R. Pellizzoni and M. Caccamo. Towards the predictable integration of real-time COTS based systems. In *Proc. of the* 28^{th} *IEEE Real-Time System Symposium*, Dec 2007.

[72] R. Pellizzoni and M. Caccamo. Impact of peripheral-processor interference on wcet analysis of real-time embedded systems. *IEEE Trans. on Computers*, 59(3):400–415, Mar 2010.

[73] R. Pellizzoni, P. Meredith, M. Caccamo, and G. Roşu. BusMOP: a runtime monitoring framework for PCI peripherals. Technical report, University of Illinois at Urbana-Champaign, 2008. Available at `http://netfiles.uiuc.edu/rpelliz2/www/`.

[74] R. Pellizzoni, P. Meredith, M. Caccamo, and G. Rosu. Hardware runtime monitoring for dependable cots-based real-time embedded systems. In *Real-Time Systems Symposium, 2008*, pages 481–491, 30 2008-Dec. 3 2008.

[75] R. Pellizzoni, P. Meredith, M. Nam, M. Sun, M. Caccamo, and L. Sha. Handling mixed criticality in soc-based real-time embedded systems. In *Proceedings of the International Conference on Embedded Software (EMSOFT)*, Grenoble, France, Oct 2009.

[76] R. Pellizzoni, A. Schranzhofer, J.-J. Chen, M. Caccamo, and L. Thiele. Worst case delay analysis for memory interference in multicore systems. In *Proceedings of Design, Automation and Test in Europe (DATE)*, Dresden, Germany, Mar 2010.

[77] PetaLogix. Petalinux. http://developer.petalogix.com/, 2008.

[78] John Pike. Hh-60g pave hawk. www.globalsecurity.org/ military/ systems/ aircraft/ hh-60g.htm, 2009.

[79] I. Puaut and D. Hardy. Predictable paging in real-time systems: A compiler approach. In *ECRTS '07: Proc. of the 19th Euromicro Conf. on Real-Time Systems*, pages 169–178, 2007.

[80] H. Ramaprasad and F. Mueller. Bounding preemption delay within data cache reference patterns for real-time tasks. In *Proc. of the IEEE Real-Time Embedded Technology and Application Symposium*, Apr 2006.

[81] J. Reineke, D. Grund, C. Berg, and R. Wilhelm. Timing predictability of cache replacement policies. *Real-Time Systems*, 37(2), 207.

[82] J. Rosen, P. Eles A. Andrei, and Z. Peng. Bus access optimization for predictable implementation of real-time applications on multiprocessor systems-on-chip. In *Proc. of the* 28^{th} *IEEE Real-Time System Symposium*, December 2007.

[83] A.-E. Rugina, K. Kanoun, and M. Kaaniche. The ADAPT Tool: From AADL Architectural Models to Stochastic Petri Nets through Model Transformation. In *Dependable Computing Conference, 2008. EDCC 2008. Seventh European*, pages 85–90, May 2008.

[84] A. Sangiovanni-Vincentelli. Quo vadis, SLD? reasoning about the trends and challenges of system level design. *Proceedings of the IEEE*, 95(3):467–506, 2007.

[85] S. Schliecker, M. Negrean, G. Nicolescu, P. Paulin, and R. Ernst. Reliable performance analysis of a multicore multithreaded system-on-chip. In *CODES/ISSS*, pages 161–166, 2008.

[86] Simon Schliecker, Mircea Negrean, and Rolf Ernst. Bounding the shared resource load for the performance analysis of multiprocessor systems. In *DATE*, Dresden, Germany, mar 2010.

[87] S. Schönberg. Impact of pci-bus load on applications in a pc architecture. In *Proceedings of the 24th IEEE International Real-Time Systems Symposium*, Cancun, Mexico, Dec 2003.

[88] Andreas Schranzhofer, Jian-Jia Chen, and Lothar Thiele. Timing analysis for TDMA arbitration in resource sharing systems. In *RTAS*, 2010.

[89] Boston Scientific. Pacemaker system specifcation. sqrl.mcmaster.ca/ _SQRLDocuments/ PACEMAKER.pdf, 2007.

[90] I. Shin and I. Lee. Periodic resource model for compositional real-time guarantees. In *Proceedings of Proceedings of the 23th IEEE Real-Time Systems Symposium*, Cancun, Mexico, Dec 2003.

[91] F. Singhoff, J. Legrand, L. Nana, and L. Marcé. Scheduling and memory requirements analysis with AADL. In *Proceedings of ACM SIGAda*, volume 25, pages 1–10, Atlanta, Georgia, 2005.

[92] O. Sokolsky, I. Lee, and D. Clarke. Schedulability analysis of AADL models. *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, April 2006.

[93] B. Sprunt, L. Sha, and J. Lehoczky. Aperiodic task scheduling for hard-real-time systems. *Journal of Real-Time Systems*, 1, July 1989.

[94] M. Spuri. Analysis of deadline scheduled real-time systems. Technical Report RR-2772, INRIA, France, January 1996.

[95] M. Spuri and G.C. Buttazzo. Efficient aperiodic service under earliest deadline scheduling. In *Real-Time Systems Symposium, 1994., Proceedings.*, pages 2–11, Dec 1994.

[96] H. Sun, M. Hauptman, and R. Lutz. Integrating product-line fault tree analysis into aadl models. In *High Assurance Systems Engineering Symposium, 2007. HASE '07. 10th IEEE*, pages 15–22, 2007.

[97] Symtavision. SymTA/S Toolbox. http://www.symtavision.com/symtas.html.

[98] Lothar Thiele, Samarjit Chakraborty, and Martin Naedele. Real-time calculus for scheduling hard real-time systems. In *International Symposium on Circuits and Systems ISCAS 2000*, volume 4, pages 101–104, Geneva, Switzerland, March 2000.

[99] K. Tindell, A. Burns, and A. Wellings. An extendible approach for analysing fixed priority hard real-time tasks. *Journal of Real Time Systems*, 6(2):133–151, Mar 1994.

[100] University of Illinois. *HW-SW Architectures for SoC-based Real-Time Embedded Systems*. http://netfiles.uiuc.edu/rpelliz2/www/soc.html.

[101] Uppsala University and Aalborg Univeristy. *Uppaal a tool suite for verification of real-time systems*. www.uppaal.com.

[102] E. Wandeler, L. Thiele, M. Verhoef, and P. Lieverse. System architecture evaluation using modular performance analysis: a case study. *International Journal on Software Tools for Technology Transfer*, 9(6):649–667, Nov 2006.

[103] John G. Webster, editor. *Design of cardiac pacemakers*. IEEE Press, Piscataway, NJ, 1995. ecow.engr.wisc.edu/ cgi-bin/ get/ bme/ 762/ webster/ designofca/.

[104] J. Whitham and N. Audsley. Implementing time-predictable load and store operations. In *Proc. of the Intl. Conf. on Embedded Systems (EMSOFT)*, Grenoble, France, Oct 2009.

[105] R. Wilhelm, D. Grund, J. Reineke, M. Schlickling, M. Pister, and C. Ferdinand. Memory hierarchies, pipelines, and buses for future architectures in time-critical embedded systems. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 28(7):966–978, Jul 2009.

[106] Xilinx. Virtex-5 LXT FPGA ML505 Evaluation Platform. www.xilinx.com/products/devkits/HW-V5-ML505-UNI-G.htm, 2008.

[107] Xilinx. Virtex-5 LXT ML555 FPGA Development Kit for PCI Express, PCI-X, and PCI Interfaces. www.xilinx.com/ products/ devkits/ HW-V5-ML555-G.htm, 2009.

[108] Xilinx, Inc. *MicroBlaze Processor Reference Guide*. http://www.xilinx.com/.

[109] Xilinx, Inc. *Virtex-4 ML455 PCI/PCI-X Development Kit User Guide*. `http://www.xilinx.com/support/documentation/boards_and_kits/ug084.pdf`.

[110] Xilinx, Inc. *Virtex-4, Virtex-II Pro and Virtex-II Pro X FPGA User Guide*. http://www.xilinx.com/.

[111] Xilinx, Inc. *Virtex-5 User Guide*. Available at www.xilinx.com.